VŠB TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Modul pro Q-Learning pro Modeler neuronových sítí

Module Q-Learning for Neuron Net Modeler

Bc. Jan Bauer

Diplomová práce

Vedoucí práce: Ing. David Ježek, Ph.D.

Ostrava, 2021

Abstrakt

Tato diplomová práce se zabývá rozšířením programu Modeler neuronových sítí o modul pro algoritmus typu učení reinforcement learning zvaný Q-learning s využitím umělých neuronových sítí. Cílem je demonstrovat tento typ učení na prostředí vybraných typů klasických arkádových her od společnosti Atari. Práce se také zaměřuje i na paralelizaci této metody učení. V této práci jsou tedy popsány předpoklady pro porozumění problematice propojení Q-learning algoritmu s umělými neuronovými sítěmi a jeho následná adaptace na přidělený typ testovacího prostředí. Dále návrh a postup k vypracování zadání této práce, způsob nastavení parametrů algoritmu, technologie použité pro implementaci a zhodnocení výsledku práce.

Klíčová slova

diplomová práce; Reinforcement learning; Deep Q-learning; Artificial neural networks; Java; JavaFX

Abstract

This thesis focuses on extension of Neural net modeler by module for one of the reinforcement learning algorithm called Q-learning with the use of artificial neural network. The aim is to demonstrate this type of learning on chosen environment types of classic arcade videogames from Atari company. Thesis is also focused on parallelization of this method of learning. In this thesis are described assumptions to understand problematics connection of Q-learning algorithm with artificial neural networks and its adaptation on assigned type of testing environment. Then the design and developing process of this thesis task, algorithm parameters setting methods, technologies used for implementation and conclusion evaluating the result.

Keywords

master thesis; Reinforcement learning; Deep Q-learning; Artificial neural networks; Java; JavaFX

Poděkování

Rád bych na tomto místě poděkoval všem dobrým lidem v mém okolí, kteří nějakým způsobem přispěli k tomu, aby tato práce vznikla.

Hlavně bych chtěl pak poděkovat Ing. Davidu Ježkovi, Ph.D. za řádné vedení této práce, pravidelné konzultace a věcné připomínky a rady.

Obsah

Seznam použitých zkratek a symbolů

ANN	–	Artificial Neural Network
DNN	–	Deep Neural Network
RNN	–	Recurrent Neural Network
RL	–	Reinforcement learning
DQN	–	Deep Q Network
MDP	–	Markov Decision Process
TD	–	Temporal difference
AI	–	Artificial inteligence
UI	–	User Interface
CSS	–	Cascading Style Sheets
XML	–	Extensible Markup Language
YAML	–	YAML Ain't Markup Language
POM	–	Project Object Model
JAR	–	Java ARchive
IT4I	–	IT4Innovations
HPC	–	High-performance computing
PBS	–	Portable Batch System
UML	–	Unified Modeling Language

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Pravděpodobně asi jedna z prvních myšlenek, která nás napadne, když přemýšlíme o povaze učení je, že nejlépe se učíme interakcí s naším prostředím. Po celý život jsou interakce nepochybně hlavním zdrojem znalostí o našem prostředí a nás samotných. Ať už se učíme řídit auto nebo vést rozhovor, jsme si vědomi toho, jak naše prostředí reaguje na to, co děláme a to, co se děje, se snažíme ovlivňovat naším chováním. Učení z interakce je tedy základní myšlenkou, která je podkladem téměř všech teorií učení a inteligence.

Pokud přemýšlíme o učení pomocí interakce z výpočetního hlediska, tak se takovýto přístup nebo typ strojového učení nazývá Reinforcement learning, který je na toto téma nejvíce orientovaný. Tento přístup je obsažen v této práci, která je zaměřena na jeden z Reinforcement learning algoritmů - Q-learning.

Dále pokud přemýšlíme o strojovém učení, tak asi nepoužívanějším výpočetním modelem, který ho reprezentuje, jsou umělé neuronové sítě, které si v posledních letech nacházejí čím dál větší uplatnění v mnoha oborech. Je to hlavně díky jejich jedinečné schopnosti extrahovat význam z nepřesných nebo složitých dat, ve kterých nachází různé vzorce a struktury, které jsou příliš náročné pro lidský mozek nebo pro jiné počítačové techniky. Asi největším průkopnickým aspektem umělých neuronových sítí je to, že se dokáží adaptativně učit. Stejně jako lidé i umělé neuronové sítě modelují nelineární a složité vztahy a staví na předchozích znalostech.

Pak může vyvstat otázka: Mohou být umělé neuronové sítě spojeny s Reinforcement learning přístupem, kdy se agent může učit nejblíže tomu, jak se učí člověk, a to interakcí s prostředím?

Tento přístup již tedy existuje a nazývá se Deep Q-learning, kdy během tréninkového procesu agent interaguje s prostředím a přijímá data, která se používají během učení umělé neuronové sítě a ta pak aproximuje jeho budoucí chování.

Hlavním obsahem této práce je právě toto propojení algoritmu Q-learning s hlubokou umělou neuronovou sítí a jeho využití demonstrované v testovacím prostředí klasických arkádových her.

Kapitola 2

Teoretický popis

2.1 Reinforcement learning

Reinforcement learning [1] nebo-li Zpětnovazebné učení je učení, jakým způsobem mapovat situace na akce, aby bylo možné co nejlépe maximalizovat numerický odměnový signál. Učícímu se agentovi není sděleno, jaké akce by měl vykonat, ale místo toho on sám musí zjistit, které akce přinášejí největší odměnu tím, že je zkouší při přímé interakci s prostředím, ve kterém se nachází. Tímto se liší od v současnosti nejběžnějších učení v rámci strojového učení tj. machine learning, a to *Supervised learning* - tedy učení se dle trénovací množiny značených dat, které jsou poskytnuty externím učitelem a *Unsupervised learning* - tedy učení se z trénovací množiny dat, které nejsou značena, takže se v nich hledají různé skryté struktury a shluky podobných dat. Dále ze všech forem strojového učení je Reinforcement learning nejbližší způsobu učení, jakým se učí lidé nebo zvířata, kdy podstatná většina algoritmů tohoto učení byla originálně inspirována biologickými systémy učení. Reinforcement learning je také silně spojen či ovlivněn psychologií nebo neurovědou např. se systémem odměn v mozku. Hlavně ale Reinforcement learning k definování interakce agenta a prostředí využívá Markovových rozhodovacích procesů (viz. dále).

2.1.1 Markovovy rozhodovací procesy

Markovovy rozhodovací procesy [2] poskytují idealizovaný matematický rámec pro modelování rozhodování v situacích, tedy volby různé akce v dané situaci, kdy jsou výsledky zčásti náhodné a zčásti pod kontrolou uživatele. Modelují tedy rozhodování v diskrétních, stochastických a sekvenčních prostředích. Využívají se pro studium mnoha typů optimalizačních problémů řešených prostřednictvím dynamického programování a Reinforcement learning učení. Jsou zde představeny klíčové elementy matematické struktury problému:

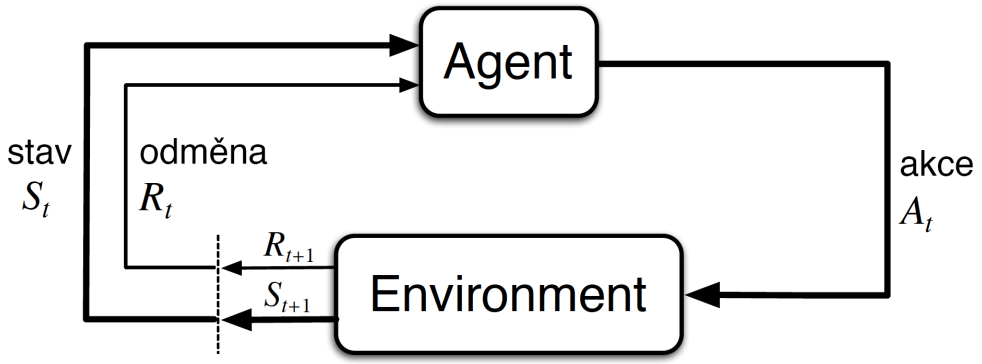
- návratová hodnota/odměna;
- hodnotová funkce;

- Bellmanova rovnice;

2.1.2 Rozhraní Agent-Environment

Markovovy rozhodovací procesy přímo modelují problém učení se z interakce k dosažení požadovaného cíle. Žák nebo entita, která se rozhoduje je nazýván *Agent*. To, s čím interaguje, zahrnující vše mimo agenta, je nazýváno *Environment (dále prostředí)*. Tato interakce probíhá tak, že si agent vybírá dané *akce tj. Action*, které vykonává v daném prostředí a prostředí mu odpovídá na zvolené akce a předkládá mu jeho novou situaci, nebo-li *stav tj. State*. Akce jsou tedy závazné a ovlivňují i stav celého prostředí např. další šachová pozice, další lokace robota, úroveň hladiny v zásobníku, apod. Prostředí mu také vrací odměny za tyto akce v podobě speciálních číselných hodnot tak, že je agent má snahu maximalizovat v průběhu jeho volby dalších akcí. Akce a stavy se reprezentují hodnotovou funkcí jako tzv. *State-Action Pair*.

Prostředí se dělí dle typu jeho modelu, a to na tzv. *model-based*, který je založen na metodě *plánování*, tedy vyhodnocení akce vzhledem k budoucí situaci, která ještě nebyla vyzkoušena - agent tedy zná model prostředí. A pak je zde tzv. *model-free*, který využívá vlastně opak plánování, a to *trial-and-error tj. pokus-omyl* metodu - agent se tedy pohybuje ve stochastickém prostředí. Tedy agent si vlastně postupně sám vytváří určitý vlastní model prostředí, ve kterém se nachází, a hledá sám co nejlepší způsob chování, aby co nejlépe mohl dosahovat daného cíle, což je asi nejlepší přístup, kterým se tato práce bude dále zabývat.



Obrázek 2.1: Interakce Agent-Enviroment v Markovových rozhodovacích procesech [1]

2.1.3 Cíle a odměny

Reinforcement learning tedy oproti jiným učením začíná s kompletním, cílevědomým agentem, který se zlepšuje pomocí *odměn tj. reward*. Agent se tedy snaží maximalizovat celkovou hodnotu odměny, kterou získá, tzn. maximalizovat nikoli okamžitou odměnu, ale celkovou odměnu v dlouhodobém horizontu. K získání co největší odměny musí agent preferovat akce, které již zkusil v minulosti a zjistil, že za ně dostává efektivní odměnu. Ovšem k objevení takovýchto efektivních akcí musí

zkoušet akce, které předtím ještě nezkusil. Agent proto musí *zužitkovat tj. exploit* zkušenosti, které již získal a zároveň musí *prozkoumávat tj. explore*, aby v budoucnu získal lepší výběr akcí. Tímto, tedy výběrem akcí se zabývají dané tzv. *politiky tj. policies*, např. Epsilon-greedy policy. [3]

2.1.4 Shrnutí klíčových elementů

- *agent* - žák/entita, která se učí a rozhoduje (tzv. decision maker);
- *prostředí/environment* - vše, s čím agent interaguje kromě agenta samotného;
- *model* - typ prostředí, ve kterém se agent nachází - *model-free* a *model-based*;
- *akce/actions* - jsou volby vykonávané agentem;
- *stavy/states* - jsou základem pro volbu akcí;
- *odměny/rewards* - slouží pro ohodnocení vykonané akce v daném stavu;
- *politika/policy* - pravidla, podle kterých si agent vybírá akce;

2.2 Q-learning algoritmus

Q-learning algoritmus [4], který byl využit v rámci tohoto projektu spadá do podkategorie učení v rámci Reinforcement learning učení tedy pod *Temporal difference learning*. Další podkategorie Reinforcement learning učení jsou *Dynamické programování* a *Monte Carlo Methods*, ale Temporal Difference learning má asi největší využití, jelikož je vlastně kombinací těchto dvou podkategorii a ve výsledku tedy nejlépe vystihuje celý Reinforcement learning.

2.2.1 Algoritmus

Q-learning spadá ještě v rámci Temporal Difference Learning [4] do kategorie tzv. *Off-policy* algoritmů. Tedy je nezávislý na dané politice, podle které by aproximoval hodnotovou state-action funkci Q a místo toho ji aproximuje přímo, ale je stále vždy dáno, které dvojice state-action jsou navštíveny a aktualizovány. Dále Q-learning nevyžaduje model prostředí je tedy *model-free*. *"Q"* označuje funkci, která vrací odměnu využitou k učení a můžeme tedy říci, že to znamená *kvalitu (quality)* zvolené akce v daném stavu.

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{stará hodnota}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Big(\underbrace{r_t}_{\text{odměna}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{odhad optimální budoucí hodnoty}} }^{\text{temporal difference}} \underbrace{- Q(s_t, a_t) \Big)}_{\text{stará hodnota}}$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{nová hodnota (temporal difference target)}}$$

Obrázek 2.2: Q-learning rovnice [5]

Q-learning rovnice na obrázku 2.2 vychází z Bellmanovy rovnice, kdy do ní byli přidány dané Q hodnoty a konstanty. Pro průběh výpočtu je potřebná tzv. *Q-table* [5] tedy tabulka Q hodnot, kde jsou zaznamenávány hodnoty state-action dvojic, které jsou vypočteny touto rovnicí po každém kroku, který agent učiní a dostane odezvu z prostředí. V Q-table každý řádek reprezentuje stav, kterému se poté pro danou akci změní její hodnota vyjádřená číselně. Tyto hodnoty jsou pak využity pro politiku výběru akcí viz. Exploration vs Exploitation. Mimo jiné lze Q-table nahradit neuronovou sítí a tím značně zlepšit výsledky a úspěchy algoritmu - jedná se o tzv. *DQN - Deep Q Neural Network*. [6]

2.2.2 Exploration vs Exploitation - výběr akcí

Jedná se o způsob výběru akcí, který agent provádí vždy v každém stavu, aby se dostal do dalšího stavu.

Exploration - dovoluje agentovi vylepšit své dosavadní poznatky o prostředí a prozkoumávat jej, tak, že z možných akcí vybere náhodně

Explotation - dovoluje agentovi využít nebo zužitkovat své již dosažené poznatky a vybere akci s nejvyšší hodnotou

K tomuto účelu slouží tzv. *Epsilon-Greedy policy* [3], která slouží k balancování exploration a explotation, tak, že mezi nimi vybírá náhodně. *Epsilon* zde vyjadřuje číselné vyjádření pravděpodobnosti výběru pro průzkum, který má menší šanci pro výběr a většinou se tato pravděpodobnost také v průběhu běhu algoritmu zmenšuje.

```
p = random()
if p < ε then
 |  vyber náhodnou akci
else
 |  vyber současnou nejlepší akci
end
```

Obrázek 2.3: Průběh Epsilon-Greedy policy [3]

2.2.3 Průběh Q-learning algoritmu

Ve výpisu na obrázku 2.4 můžeme vidět, jak celý průběh algoritmu vypadá se zahrnutím všech již zmíněných pojmů a technik.

Inicializace algoritmu a Q table
Inicializace počátečního ϵ
Inicializace Agenta a testovacího prostředí Environment
while *nejsou splněny ukončovací parametry* **do**
    Inicializace počátečního stavu s
    for *pro každý krok v epizodě* **do**
        Výběr vhodné akce a pro současný stav s dle Epsilon-Greedy policy
        Vykonání vybrané akce a Agentem a následné získání odměny r a nového stavu prostředí s'
        Výpočet Q hodnoty dle předchozího stavu s, zvolené akce a, současného nového stavu s'
        Vložení vypočtené Q hodnoty do Q table
    end
    Dekrementace ϵ a aktualizace parametrů
end

Obrázek 2.4: Průběh Q-learning algoritmu [1]

2.3 Umělá neuronová síť

Umělá neuronová síť (dále ANN - Artificial Neural Network) [6] je biologicky inspirovaný výpočetní model navržený k simulaci způsobu, jakým lidský mozek analyzuje a zpracovává informace. Tyto sítě jsou tedy budovány podobně jako lidský mozek s propojenými neuronovými uzly tvořící síť. Lidský mozek má stovky miliard buněk zvaných neurony. Každý neuron je tvořen buněčným tělem, které je odpovědné za zpracování a přenášení informací.

Neurony jsou tedy takovými procesory signálu a můžeme si pak představit, že neuron má ve vstupech sběrač signálu a aktivační jednotku na výstupu, která může spustit signál, který bude předán jiným neuronům. Spojené neurony jsou pak reprezentovány váhami, které představují spojení mezi neurony a mají schopnost zesílit nebo zeslabit neuronové signály, například znásobit signály, a tím je upravovat.

Tak i ANN se skládá ze síťové architektury složené z až stovek nebo tisíců umělých neuronů, které reprezentují procesní jednotky jako uzly, které jsou také tvořeny vstupními a výstupními částmi a jsou vzájemně propojeny tzv. váhami v podobě hran. Tyto umělé neurony jsou organizovány do vrstev, které tvoří model ANN. Poté po průchodu tímto síťovým modelem ANN je pro daný vstup, reprezentovaný vstupní vrstvou umělých neuronů, vrácen daný požadovaný výstup. ANN také používá k zdokonalení svých výstupních výsledků sadu pravidel učení, kterou představuje daný optimalizační algoritmus např. Stochastic gradient descent s využitím metody zpětného šíření nebo-li Back-Propagation.

Základní a podstatná vlastnost ANN je učení. Jeho průběh nejprve prochází fází školení, kde se naučí rozpoznávat vzorce v datech, ať už vizuálně, sluchově nebo textově. Během této kontrolované fáze síť porovnává svůj skutečný produkovaný výstup s tím, co měl produkovat - požadovaný výstup. Rozdíl mezi oběma výsledky je upraven pomocí zpětného šíření. To znamená, že síť pracuje pozpátku, přechází od výstupní jednotky ke vstupní jednotce, aby upravila váhu svých spojení mezi jednotkami, dokud rozdíl mezi skutečným a požadovaným výsledkem nevytvoří nejnižší možnou chybu. Dále ANN využívají distribuované, paralelní zpracování informace při provádění výpočtů. Znalosti jsou ukládány především prostřednictvím síly vazeb mezi jednotlivými neurony. Vazby mezi neurony vedoucí ke "správné odpovědi"jsou posilovány a naopak, vazby vedoucí k "špatné odpovědi"jsou oslabovány pomocí opakované expozice příkladů popisujících problémový prostor. Neuronové sítě se tedy učí úpravou spojení mezi neurony, a to úpravou vah, jejichž přizpůsobením dle pravidel učení může ANN zlepšit své výsledky.

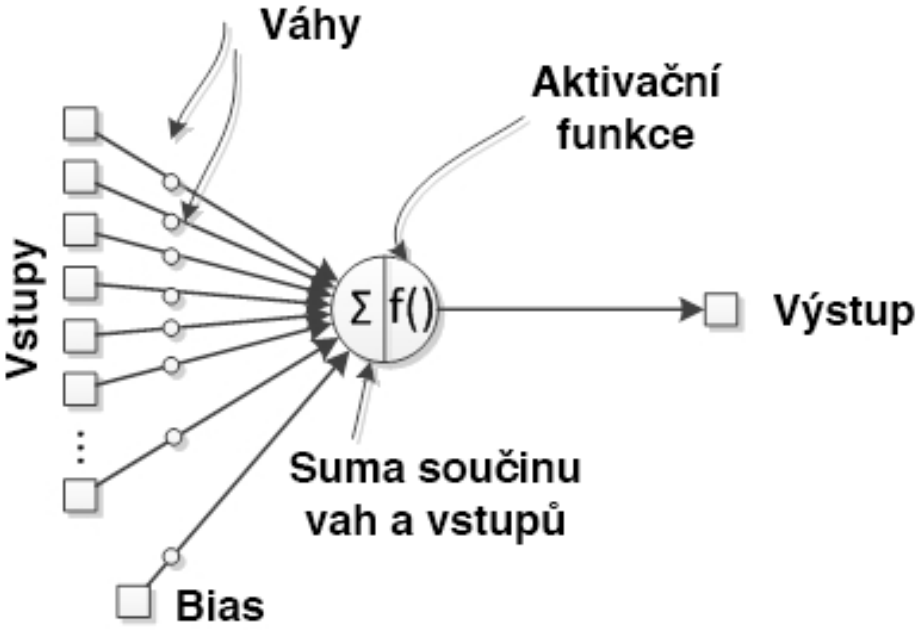
ANN může být „shallow“, což znamená, že má vstupní vrstvu neuronů a pouze jednu skrytou vrstvu (hidden layer), která zpracovává vstupy, a výstupní vrstvu, která poskytuje konečný výstup modelu. Opakem je „deep“ neuronová síť (DNN), která má obvykle mezi 2–8 skrytými vrstvami neuronů.

ANN je tedy vlastně základem umělé inteligence (AI) a řeší problémy, které by se podle lidských nebo statistických standardů ukázaly jako nemožné nebo obtížné. Oblasti využití ANN zahrnují

identifikaci a řízení systémů (řízení vozidla, predikce trajektorie, řízení procesů, správa přírodních zdrojů), kvantovou chemii, obecné hraní her, rozpoznávání vzorů (radarové systémy, identifikace tváře, klasifikace signálů, 3D rekonstrukce, rozpoznávání objektů a další), rozpoznávání sekvencí (gesta, řeči, rozpoznávání ručně psaného a tištěného textu), lékařská diagnostika, finančnictví (např. Automatické obchodní systémy), dolování dat, vizualizaci, strojový překlad , filtrování sociálních sítí a filtrování spamu e-mailů, aj.

2.3.1 Hlavní komponenty umělých neuronových sítí

- *Neuron* - základní element ANN, který konceptuálně vychází z biologického neuronu. Každý umělý neuron má vstupy v podobě vah a produkuje jediný výstup, který lze odeslat do několika dalších neuronů. Každý umělý neuron je charakterizován svými vstupními váhami, prahovou hodnotou (bias), sum funkcí a aktivační funkcí.

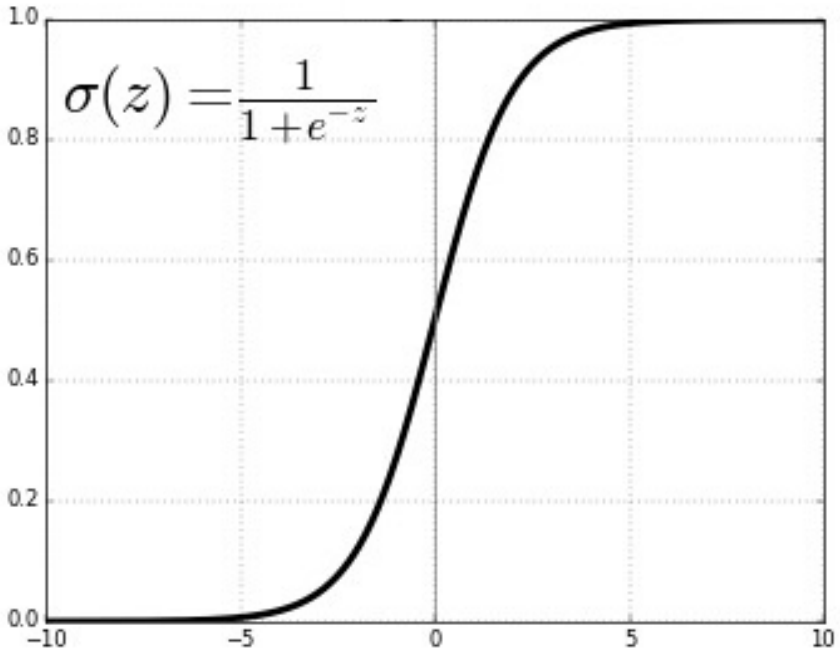


Obrázek 2.5: Struktura umělého neuronu [7]

- *Váha* - Váhy představují spojení mezi neurony. Můžeme je považovat za znalost neuronové sítě s tím, že změna vah změní i schopnosti neuronové sítě a tedy i akce.

- *Prahová hodnota - Bias* - Bias je proměnná, která pomáhá modelu takovým způsobem, aby nejlépe vyhovoval daným datům. Umožňuje posunout a tím optimalizovat aktivační funkci doleva nebo doprava, což může být rozhodující pro úspěšné učení.

- *Sum funkce* - Sum funkce spojuje více vstupních signálů neuronu do jedné skalární hodnoty, aby bylo možné použít aktivační funkci. To se obvykle provádí součtem všech vstupních signálů, které neuron přijímá, přičemž každý vstupní signál je vynásoben příslušnými váhami na připojeních.
- *Aktivační funkce* - Aktivační funkce jsou matematické rovnice, které určují výstup každého neuronu a tedy celkově i celé neuronové sítě. Jejich vstupem je skalární výstup ze sum funkce. Na základě toho, zda je vstup každého neuronu relevantní pro predikci modelu, určuje, zda má být aktivován či nikoli. Aktivační funkce také pomáhají normalizovat výstup každého neuronu na rozmezí mezi 1 a 0 nebo mezi -1 a 1 určují výstup neuronu. Nejčastěji používanou funkcí je Sigmoid (Logistická funkce) kvůli jeho snadno odlišitelným vlastnostem, což je velmi výhodné při použití Back-Propagation algoritmu. Další aktivační funkce jsou např. TanH, ReLU, Softmax, aj.

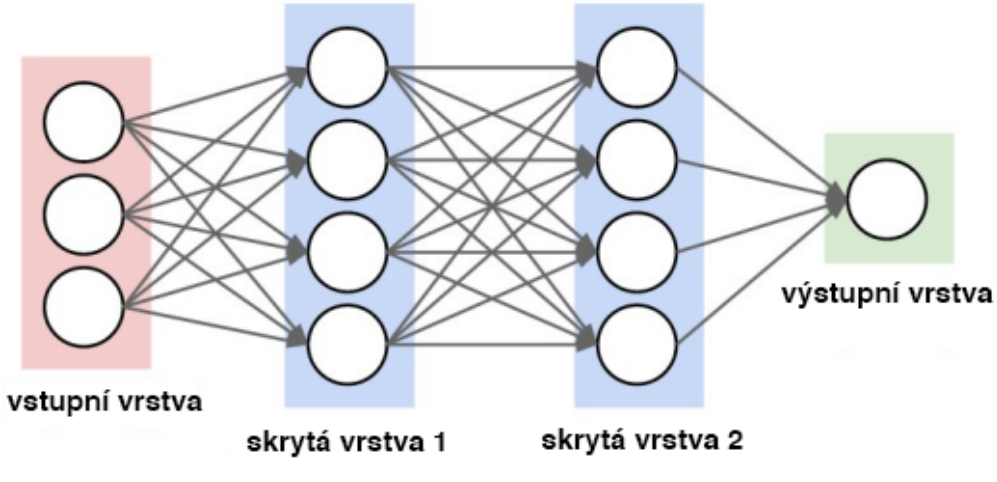


Obrázek 2.6: Sigmoid funkce [6]

- *Vrstvy (Layers)* - se dělí do 3 typů:
  - *Vstupní vrstva (Input layer)* - Vstupní vrstva se stará jenom o vstupy ANN, kdy komunikuje s externím prostředím, které by mělo představovat stav, pro který ANN trénujeme. Každý vstupní neuron by měl představovat nějakou nezávislou proměnnou, která má vliv na výstup ANN.

– *Skrytá vrstva (Hidden layer)* - Skrytá vrstva je kolekce neuronů, na které je aplikována aktivační funkce a je to mezivrstva mezi vstupní vrstvou a výstupní vrstvou. Jejím úkolem je zpracovávat vstupy získané předchozí vrstvou. Je to tedy vrstva, která je zodpovědná za extrahování požadovaných funkcí ze vstupních dat. V ANN může být také většinou několik skrytých vrstev.

– *Výstupní vrstva (Output layer)* - Výstupní vrstva je výstupem nebo výsledkem ANN, kdy shromažďuje a přenáší informace odpovídajícím způsobem tak, jak bylo navrženo pro získání požadovaných výstupů ANN pro daný problém. Počet neuronů ve výstupní vrstvě by měl přímo souviset s typem práce, kterou je ANN určena.



Obrázek 2.7: Vrstvy ANN [8]

2.3.2 Učení umělých neuronových sítí

- *Loss/Cost/Error funkce* - Funkce, která slouží výpočtu chyby ANN, což se potom využije v rámci učení k úpravě vah ANN. Udává tedy rozdíl mezi odhadovanými a skutečnými hodnotami na výstupu sítě. Např. Mean Squared Error (MSE), Binary Crossentropy (BCE), Categorical Crossentropy (CC), aj.

- *Learning rate* - Learning rate je koeficient, který se používá v rámci Back-Propagation k úpravě vah během učení. Definuje tak míru přizpůsobení sítě, které model provede, aby upravil chyby v každém pozorování. Malé hodnoty learning rate prodlužují dobu učení, ale jsou potenciálně přesnější, jelikož mají tendenci snižovat šanci na překročení optimálního řešení. Zároveň zvyšují pravděpodobnost uvíznutí v místních minimech. Velké hodnoty rychlosti učení mohou síť trénovat rychleji, ale mohou mít za následek, že k učení vůbec nedojde.

- *Back-Propagation* - Back-Propagation nebo-li Zpětné šíření je standardní metoda učení ANN, která se stará o úpravu vah tak, aby kompenzovala každou chybu nalezenou během učení,

tak, že je její míra efektivně rozdělena mezi váhy. Název Zpětné šíření je dán proto, protože chyba se po síti šíří zpět od výstupní vrstvy po vstupní a všechny váhy se odpovídajícím způsobem aktualizují. K úpravě vah se používá optimalizační algoritmus gradient descent nebo jeho varianta stochastic gradient descent. Tento algoritmus se stará o adaptaci dané váhy sítě dle předchozí vypočtené derivace loss funkce s ohledem na danou váhu sítě, kdy se výpočet gradientu (derivátu) pro danou váhu provádí podle řetízkového pravidla (chain rule) a tento gradient se pak využije pro účely adaptace dané váhy.

2.3.3 Paradigmata učení umělých neuronových sítí

- *Supervised learning* - Supervised learning nebo tzv. učení s učitelem je typ učení, který k učení využívá tréninkovou sadu značených dat k získání požadovaného výstupu. Tato datová sada pro školení zahrnuje vstupy a správné výstupy, které umožňují modelu učit se v průběhu času. Algoritmus měří svou přesnost pomocí loss funkce a upravuje se, dokud není chyba dostatečně minimalizována.

- *Unsupervised learning* - Unsupervised learning je typ učení, který k učení využívá tréninkovou sadu neoznačených dat, takže pro dané vstupy neexistují požadované výstupy. Cílem toho typu učení může být objevení různých skrytých struktur a vzorců nebo shluky podobných dat.

2.3.4 Hlavní typy umělých neuronových sítí

- *FeedForward (Dopředné) sítě* [6] - Dopředné neuronové sítě jsou ANN, kde spojení mezi umělými neurony netvoří cyklus. Tyto sítě byly prvním vynalezeným typem ANN. Nazývají se Dopředné, protože informace putují v síti pouze vpřed (bez smyček), nejprve přes vstupní uzly, poté skrz skryté uzly (jsou-li k dispozici) a nakonec přes výstupní uzly. Nejjednodušším typem této FeedForward neuronové sítě je *perceptron*, který má pouze vstupní vrstvu a výstupní vrstvu, tvoří ji tedy vlastně jeden uměly neuron. Druhým typem je *multi-layer perceptron (MLP)*, což je ANN složená z mnoha perceptronů a obsahuje skryté vrstvy.

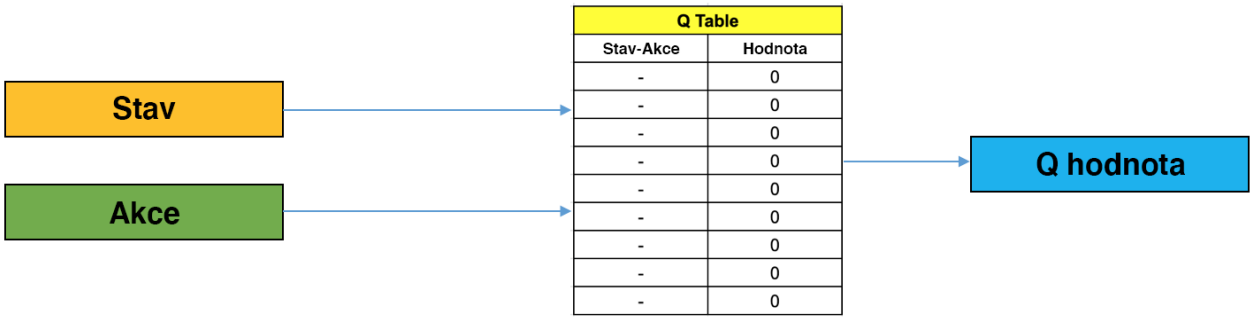
- *Recurrent (Rekurentní) sítě* [6] - Rekurentní neuronové sítě (dále RNN) jsou typem neuronové sítě, kde se výstup z předchozího časového kroku přivádí jako vstup do aktuálního časového kroku. Proto se v tomto případě informace nešíří pouze od vstupní vrstvy směrem k výstupní vrstvě, ale dochází i ke zpětnovazebnému přenosu informace od vrstev vyšších zpět do vrstev nižších. Tato zpětná vazba je realizována prostřednictvím tzv. *rekurentních neuronů*. Tyto sítě jsou pak použitelné pro úkoly, jako je jazykový překlad, zpracování přirozeného jazyka (nlp), rozpoznávání řeči a titulky obrázků, apod. Speciálním typem RNN jsou Long Short Term Memory (LSTM) sítě, které jsou schopné se učit v rámci dlouhodobých závislostí a řeší většinu běžných problémů klasických RNN.

2.4 Deep Q-learning algoritmus

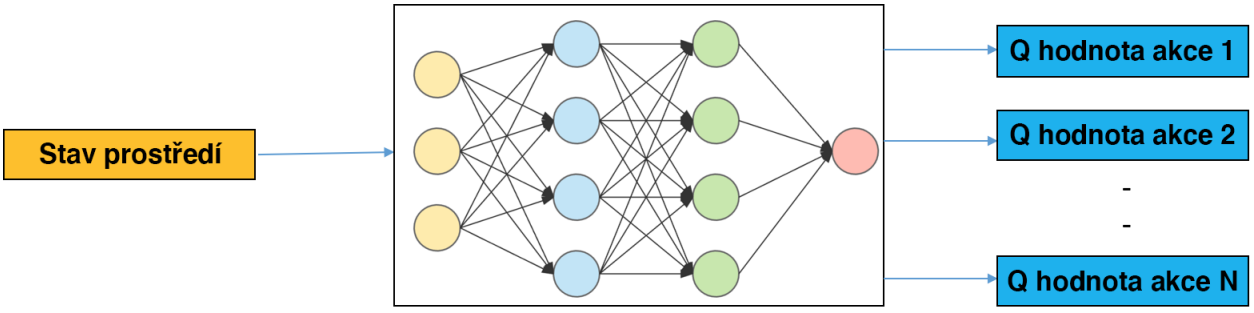
Deep Q-learning [9], představuje kombinaci Q-learning algoritmu a hluboké ANN - deep neural network (dále DNN) a neuronová síť, která potom aproximuje Q hodnoty, se tedy nazývá Deep Q-Network (dále DQN). Znamená to, že DNN se použije pro aproximaci optimální Q hodnoty daného stavu prostředí.

Základním pracovním krokem DQN pak je, že stav daného prostředí je dán jako vstup pro DNN a ta pak vrací Q hodnotu všech možných akcí jako výstup, ze něhož se ještě vybere vhodná akce dle Epsilon-Greedy policy [3]. DQN pak představuje řešení pro prostředí, které mají velký počet stavů a akcí, a tak se stávají těžko řešitelné klasickým Q-learning algoritmem.

Hlavní rozdíl mezi Q-learning algoritmem Deep Q-learning algoritmem lze ilustrovat pomocí obrázku 2.8 a obrázku 2.9:



Obrázek 2.8: Q-learning využívající Q-table [10]



Obrázek 2.9: Deep Q-learning využívající hlubokou neuronovou síť [10]

Dále se pak musí upravit Q-learning nebo spíše Bellmanova rovnice pro výpočet skutečných nebo referenčních hodnot výstupů k použití v rámci loss funkce, která se potom využije k učení DQN pomocí metody Back-Propagation.

$$Loss = (r + \gamma max_{a'} Q(s', a'; \theta') - Q(s, a; \theta))^2$$

Obrázek 2.10: Q loss funkce pro DQN [11]

2.4.1 Experience replay

Experience Replay je technika, která se v rámci DQN používá při učení. V rámci ní ukládáme zkušenosti (experiences) agenta v každém časovém kroku do trénovací množiny nebo-li do tzv. *Replay memory*, kde jsou sdruženy zkušenosti z mnoha epizod. Z Replay memory se potom vždy náhodně vybere malá dávka nebo vzorek zkušeností o dané velikosti tzv. *minibatch*, která se použije pro učení DQN. Tím se řeší problém autokorelace vedoucí k nestabilnímu tréninku tím, že se poté tento problém podobá způsobu učení metodou supervised learning.

V čase t, je agentova zkušenost e_t definována jako n-tice:

$$e_t = (s_t, a_t, r_{t+1}, s_{t+1}, done)$$

Tato n-tice obsahuje stav prostředí s_t, akci a_t vykonanou ve stavu s_t, odměnu r_{t+1} danou agentovi v čase $t+1$ za předchozí state-action dvojici (s_t, a_t), následující stav prostředí s_{t+1} a *done* znamenající zda je daný stav s_t terminální.

2.4.2 Policy a Target network

V rámci toho, že stavy s_t a $s_t + 1$ mají mezi sebou jen jeden krok, tak jsou díky tomu jsou velmi podobné a pro DQN je proto velmi těžké mezi nimi rozlišovat. Když se pak při učení DQN provádí aktualizaci parametrů, aby se hodnota $Q(s_t, a_t)$ přiblížila požadovanému výsledku, může se nepřímo změnit hodnota produkovaná pro $Q(s_t + 1, a_t + 1)$ a další blízké stavy. Díky tomu může být trénink velmi nestabilní. Proto, aby byl výcvik stabilnější, existuje způsob zvaný tzv. Target network, pomocí kterého se vždy za daný časový úsek ponechá kopie DQN, která se použije pro hodnotu $Q(s_t, a_t)$ v Bellmanově rovnici.

Takto se proces učení DQN rozdělí na tzv. Policy network a Target network:

- *Policy network* - představuje hlavní DQN, která je použita k aproximaci Q hodnot akcí a je následně trénována pro stav s_t s použitím Back-Propagation;
- *Target network* - představuje kopii Policy network, která je použita získání target hodnoty stavu s_t+1, což se využije v rámci učení Policy network. Není trénována a po určitém časovém úsek je opět zkopírována z Policy network;

2.4.3 Průběh DQN algoritmu

Základní průběh DQN můžeme shrnout ve výpisu na obrázku 2.11.

Inicializace Deep Q Policy network Q
Inicializace Deep Q Target network Q'
Inicializace Replay memory D
Inicializace počátečního ϵ
Inicializace Agenta a testovacího prostředí
while *nejsou splněny ukončovací parametry* **do**
  Aproximace možných akcí a ve stavu s pomocí Q
  Výběr vhodné akce a pro stav s dle Epsilon-Greedy policy
  Vykonání vybrané akce a Agentem a následné získání odměny r a nového stavu prostředí s'
  Uložení zkušeností $(s, a, r, s', done)$ do D
  if *dostatek zkušeností v* D **then**
    Výběr náhodné dávky zkušeností velikost N z D
    for *každou zkušenost* $(s_i, a_i, r_i, s'_i, done_i)$ **in** *dávce zkušeností* **do**
      if $done_i$ **then**
        $y_i = r_i$
      else
        $y_i = r_i + \gamma max'_a \in Q'(s', a')$
      end
      Výpočet loss funkce $L = 1/N \sum_{i=0}^{N-1}(Q(s_i, a_i) - y_i)^2$
      Aktualizace vah Q s použitím metody Back-Propagation
      Každý krok C vytvořit kopii Q' z Q
    end
  else
  end
  Dekrementace ϵ
end

Obrázek 2.11: Průběh DQN algoritmu

2.5 Atari hry - testovací prostředí

Atari, Inc [12] je součástí společnosti Infogrames, producenta počítačových her, která pod tímto názvem vystupuje na americkém trhu. Atari zahájila svoji činnost vývojem a výrobou prvních arkádových videoher. Původní společnost Atari, Inc., založená v Sunnyvale v Kalifornii v roce 1972 Nolanem Bushnellem a Tedem Dabneyem, byla průkopníkem arkádových her, domácích videoherních konzolí a domácích počítačů. Produkty společnosti, jako jsou Pong a Atari 2600, pomohly definovat průmysl elektronické zábavy od 70. do poloviny 80. let.

2.5.1 Breakout

Breakout [13] je arkádová hra vyvinutá a publikovaná společností Atari, Inc. a vydána 13. května 1976. Za konceptem stojí Nolan Bushnell a Steve Bristow a byla vytvořena Stevem Wozniakem. V roce 1978 byla hra přenesena na Atari 2600 a bylo vytvořeno pokračování Super Breakout, které se o čtyři roky později stalo „pack-in hrou“ pro konzoli Atari 5200. Breakout vytvořil celý žánr klonů a jeho koncept byl rozšířen Arkanoidem z roku 1986, který sám vytvořil desítky napodobitelů.

V Breakoutu vrstva cihel lemuje horní třetinu obrazovky a cílem je všechny je zničit. Míč se pohybuje různými směry po obrazovce a odráží se od horní a dvou stran obrazovky. Když je cihla zasažena, míč se odrazí zpět a cihla je zničena. Hráč ztratí tah, když se míč dotkne spodní části obrazovky. Aby se tomu zabránilo, má hráč vodorovně pohyblivé pádlo, které míč odrazí nahoru a udržuje jej ve hře.

2.5.2 Pong

Pong [14] je arkádová videohra s motivem stolního tenisu, která obsahuje jednoduchou dvourozměrnou grafiku, byla vytvořena společností Atari nebo specificky Allanem Alcornem a vydána byla původně v roce 1972. Je to jedna z prvních arkádových videoher a asi nejúspěšnější hra od Atari. Brzy po jeho vydání začalo několik společností vyrábět hry, které jeho hraní úzce napodobovaly.

Z hlediska hratelnosti je Pong dvourozměrná sportovní hra, která simuluje stolní tenis. Hráč ovládá herní pádlo svislým pohybem po levé nebo pravé straně obrazovky. Mohou soutěžit s jiným hráčem ovládajícím druhé pádlo na opačné straně. Hráči používají pádla k zasažení míče tam a zpět. Cílem je, aby každý hráč dosáhl jedenáct bodů před soupeřem; body se získávají, když jeden nevrátí míč druhému.

2.6 Paralelizace umělých neuronových sítí

Z hlediska zefektivnění práce hlubokých umělých neuronových sítí slouží hlavně jejich paralelizace, pomocí které je možné souběžné zpracovávání poskytnuté datové sady. Tyto paralelní výpočty se pak mohou realizovat pomocí víceprocesorových systémů nebo spuštěním úlohy na distribuovaných výpočetních uzlech, kde jsou jednotlivé počítače propojené pomocí výpočetní sítě, což je tedy v rámci zadání této práce podmínkou. Při implementaci paralelizace je pak také důležité dbát na problémy, které mohou v případě těchto výpočtů nastat. Jedná se o problémy: souběhu, synchronizace nebo paralelního zpomalení.

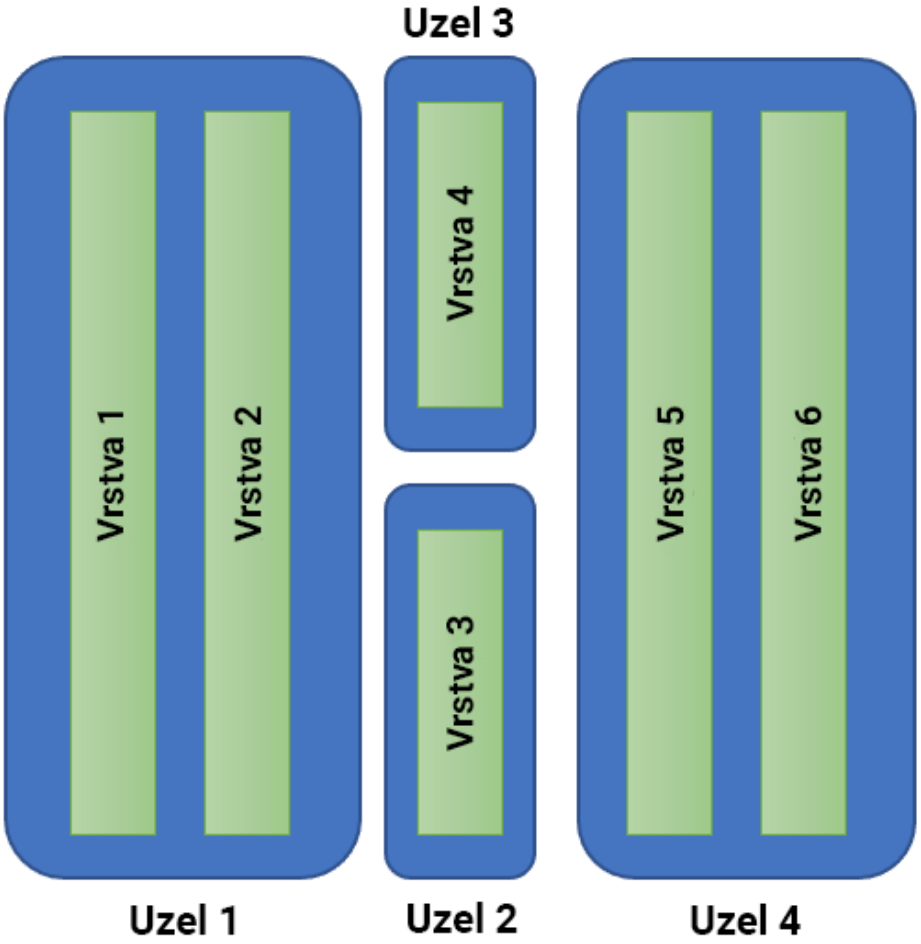
2.6.1 Dostupné popsané přístupy k paralelizaci z hlediska umělých neuronových sítí

V případě distribuovaných paralelních výpočtů z hlediska hlubokých umělých neuronových sítí se jedná hlavně o dva hlavní způsoby. A to o Modelový paralelismus a Datový paralelismus.

2.6.1.1 Modelový paralelismus

Modelový paralelismus [15] v případě DNN představuje jednu velkou síť, která rozprostřena mezi všechny uzly zahrnuté do paralelizace. A poté se tedy jednotlivé stroje/uzly starají o jednotlivé částí této sítě. Proto hlavní myšlenkou je, že je možné paralelizovat jednotlivé vrstvy nebo jednotlivé skupiny neuronů.

Tento přístup je ovšem obtížné použít, jelikož moc nepočítá se vzájemnými závislostmi mezi vrstvami či mezi skupinami neuronů. Nelze proto jednoduše provádět nezávislý paralelní výpočet, protože potom dané uzly s přidělenou částí DNN musí vždy čekat na výsledky z ostatních vrstev a vzniká tak také zbytečná komunikace mezi vrstvami DNN. Tento problém lze určitým způsobem řešit pomocí např. zavedením redundantních výpočtů do DNN nebo využitím Cannonova algoritmu pro násobení matic. Jinak tento přístup přináší určité výhody z hlediska škálovatelnosti pro větší DNN a také z hlediska toho, že nemusí určitým globálním způsobem synchronizovat dané parametry DNN po zpracování výpočetního cyklu.



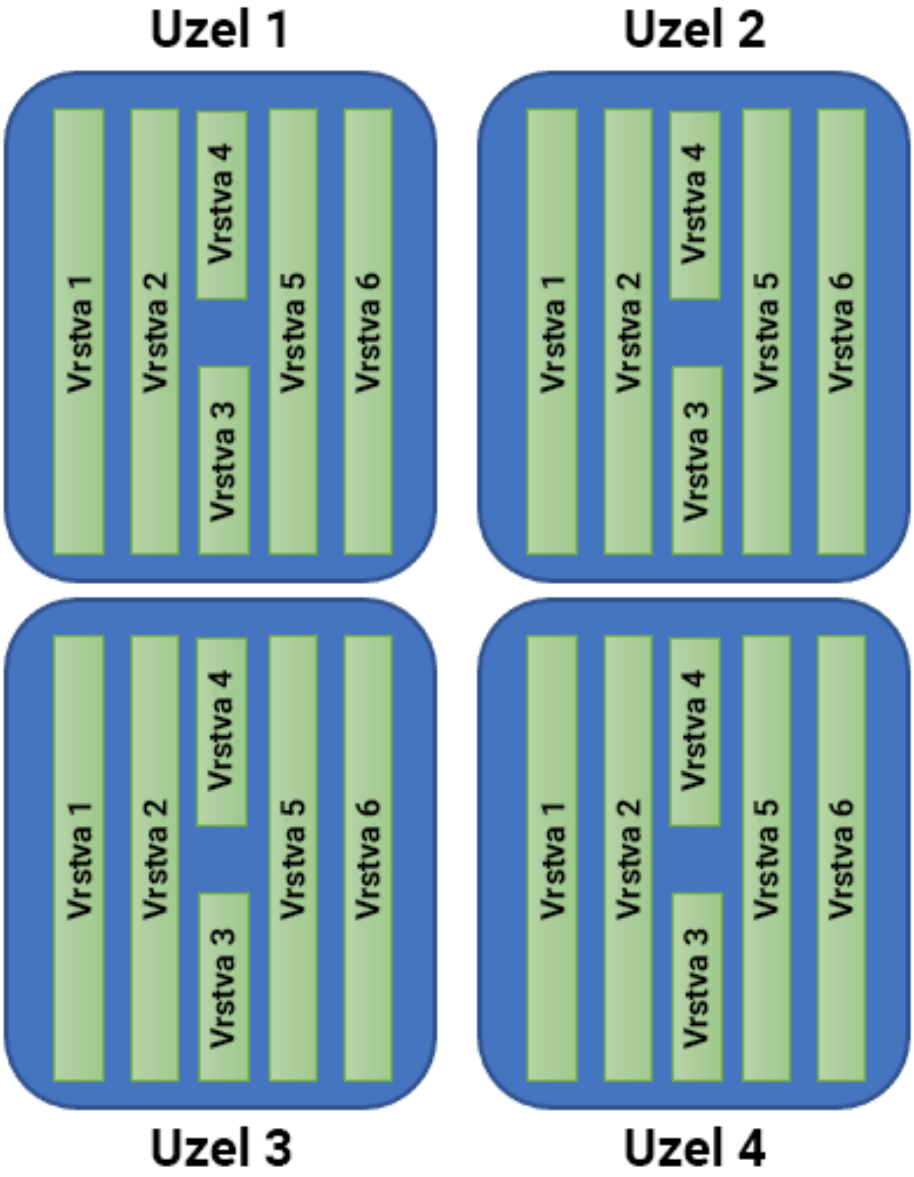
Obrázek 2.12: Modelový paralelismus [15]

2.6.1.2 Datový paralelismus

Datový paralelismus [15] na rozdíl od modelového paralelismu pohlíží na celou DNN jako na skupinu nezávislých částí. V datovém paralelismu má pak každý uzel svou vlastní kopii DNN a data pro výpočet se mezi tyto uzly rovnoměrně rozdělují. Po zpracování jednoho výpočetního cyklu všech rozdělených dávek jsou pak parametry DNN jednotlivých uzlů celkově synchronizovány do jednotné DNN, která je při začátku dalšího výpočtu rozeslána zpět na jednotlivé uzly.

Díky tomuto principu, kdy každý uzel obsahuje vlastní kopii DNN, není pak problém paralelismus realizovat. Datový paralelismus tedy nemusí řešit závislosti mezi vrstvami DNN a vyhýbá se tak zbytečné komunikaci mezi vrstvami, jak je to právě v případě modelového paralelismu. Na druhou stranu pak musí najít efektivní způsob, jak co nejlépe distribuovat data potřebné pro výpočty jednotlivým uzlům a pak také efektivně synchronizovat všechny spočtené data. Rychlost samotného výpočtu tedy závisí spíše na rychlé komunikaci mezi jednotlivými uzly, než na rychlosti výpočtu

v rámci uzlu a je tak důležité zvolit takový komunikační protokol, který je dostatečně rychlý a spolehlivý. Tento přístup se jinak používá hlavně v případě DNN s velkými trénovacími množinami.



Obrázek 2.13: Datový paralelismus [15]

Kapitola 3

Návrh

V této kapitole je popsán postupný návrh částí projektu, který se snaží navrhnout potřebné třídy dle poznatků získaných z předchozí teoretické analýzy.

Návrh zahrnuje samotný Q-learning algoritmus, umělou neuronovou síť založenou na metodě Back-Propagation, propojení Q-learning algoritmu a ANN v podobě DQN, dále prostředí pro testování a experimenty a nakonec paralelizaci tohoto způsobu učení. Jsou zde také popsány technologie, knihovny a nástroje potřebné pro samotnou implementaci.

3.1 Zvolené technologie, knihovny a nástroje

Pro implementaci projektu bylo nejprve nutné vybrat následující technologie, nástroje a knihovny, které jsou pro tyto účely vhodné a navzájem kompatibilní.

3.1.1 Programovací jazyk Java

Jako programovací jazyk pro implementaci byl vybrán jazyk Java [16].

Jedná se o programovací jazyk a výpočetní platformu, která byla poprvé vydána společností Sun Microsystems v roce 1995 a pak se po akvizici této společnosti v roce 2010 se stala vlastnictvím společnosti Oracle Corporation, která pokračuje v jejím dalším vývoji.

Tento jazyk byl vybrán hlavně z toho důvodu, že již existující projekt Modeler neuronových sítí je na tomto jazyce postaven a také je možné ho spustit v rámci HPC na výpočetních uzlech národního superpočítačového centra IT4Innovations. Dále pak proto, že je to objektově orientovaný programovací jazyk postavený na třídách, který je robustní, rychlý, bezpečný, výkonný, víceúlohový, dynamický, nezávislý na architektuře, přenositelný a realizuje generační správu paměti pomocí automatického garbage collectoru, který automaticky vyhledává již nepoužívané části paměti a uvolňuje je pro další použití. Právě díky své přenositelnosti lze zkompilovaný kód Java tzv. bytecode spustit na všech platformách, které mají k dispozici interpret Javy, tzv. Java Virtual Machine (JVM), aniž by bylo nutné ho znovu kompilovat. Tento princip se nazývá tzv. Write once, run anywhere (WORA)

- napiš jednou, spusť kdekoliv. Díky těmto vlastnostem tedy zcela splňuje všechny potřebné parametry pro tento projekt.

V rámci implementace byla pak použita verze Java SE 8 LTS, jelikož je také použita v existujícím projektu Modeler neuronových sítí a navíc ještě na rozdíl od dalších verzí nativně obsahuje grafickou knihovnu JavaFX, která je zmíněna dále.

3.1.2 Vizualizace pomocí JavaFX

JavaFX [17] je moderní knihovna pro tvorbu komplexnějších vizuálních okenních aplikací v programovacím jazyce Java. JavaFX přináší podporu obrázků, videa, animací, hudby, grafů, CSS stylů a dalších technologií. Zároveň je kladen důraz na jednoduchost tvorby, všechny zmiňované věci jsou v JavaFX v základu. V současnosti ji už vyvíjí open-source komunita openjfx.io a od Javy verze 11 není součástí JRE/JDK.

Protože tedy úzce souvisí s programovacím jazykem Java a zahrnuje všechny potřebné parametry kladené na grafickou knihovnu, tak byla vybrána pro implementaci grafického testovacího prostředí arkádových her společnosti Atari - Breakout a Pong.

3.1.3 Apache Maven

Apache Maven [18] je nástroj určený pro správu závislostí a automatizaci sestavování zejména pro Java projekty. Základem je Project Object Model (POM), což je XML soubor pom.xml, jenž popisuje vytvářený softwarový projekt, jeho závislosti na jiných externích modulech a komponentách, pořadí sestavení, adresáře a požadované pluginy.

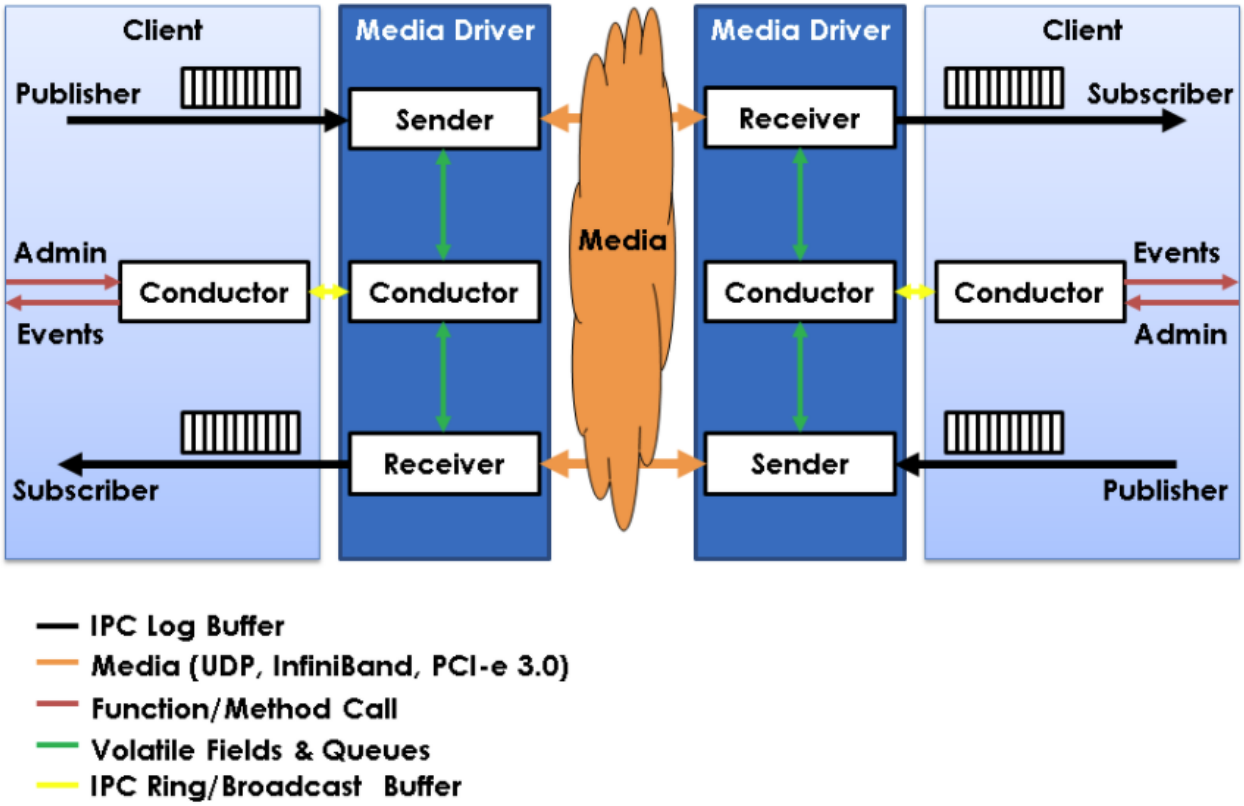
V rámci projektu této diplomové práce byl tento nástroj hlavně využit k celkovému sestavení aplikace s využitím pluginu Apache Maven Shade Plugin, který poskytuje schopnost zabalit artefakt do tzv. uber-jar spustitelných souborů, včetně všech jeho závislostí a také přejmenovat balíčky některých závislostí. Dále pak kvůli získání a použití balíčků knihoven Aeron pro síťovou komunikaci a knihovny SnakeYAML pro serializaci Java objektů do YAML dokumentů a naopak, kvůli následné konfiguraci sestavené aplikace.

3.1.4 Komunikační knihovna Aeron

Aeron [19] je síťová komunikační knihovna od společnosti Real Logic, jenž poskytuje efektivní spolehlivý UDP unicast, UDP multicast a IPC přenos zpráv. Je dostupná pro programovací jazyky Java a C++ a dále je pak třetí stranou poskytována pro .NET. Cílem Aeronu v oblasti designu je být nejvyšší propustností s co nejnižší a nejpředvídatelnější latencí jakéhokoli systému zasílání zpráv. Aeron dále integruje Simple Binary Encoding (SBE) pro nejlepší možný výkon kódování a dekódování zpráv.

Aeron je tedy transportní protokol, který navržen tak, aby běžel přímo na mnoha různých typech přenosových médií, včetně sdílené paměti/IPC, InfiniBand/RDMA, UDP, TCP, Raw IP,

HTTP, WebSocket, BLE, atd. Může tak vlasntně fungovat na nespolehlivých médiích a zároveň zajistit spolehlivý proud orientovaný na připojení.



Obrázek 3.1: Architektura síťové komunikační knihovny Aeron [19]

Díky těmto vlastnostem byl ideální volbou pro síťovou paralelizaci jak lokálně, tak i na distribuovaných výpočetních uzlech.

3.2 Návrh učení

Z hlediska učení typu Reinforcement learning [4] je, jak už bylo zmíněno dříve, hlavním rysem to, že je dán agent, který interaguje s prostředím, ve kterém se nachází. Agent sleduje aktuální stav prostředí, provádí akce, dostává odměnu za dané akce a prostředí přechází do dalšího stavu. Tento proces se opakuje, dokud není splněno určité ukončovací kritérium. Dávka stavu, akce a odměny tak tvoří jeden cyklus prostředí. Cílem agenta je pak maximalizovat jeho celkovou očekávanou odměnu získanou v jednom cyklu prostředí.

Proto, aby agent mohl ještě lépe a efektivněji tuto odměnu maximalizovat, nám poslouží ANN, která bude sloužit k uchování zkušeností v rámci svých vah a bude pak aproximovat optimální Q hodnoty akcí daného stavu prostředí. Poté je třeba definovat samotné propojení ANN s Q-learning

algoritmem. Dále aplikovat určité techniky a přístupy, aby učení bylo co nejstabilnější. Nakonec pak vytvořit takové prostředí, ve kterém agent bude moci ovládat danou entitu vykonávající možné akce, za které mu prostředí bude vracet patřičnou odměnu, kterou si potom zpracuje v rámci své ANN s využitím metody Back-Propagation.

Vše by mělo být schopno běžet jak v grafickém režimu, kde bude učení probíhat přímo v reálném čase, tak i čistě na pozadí v konzolové verzi, kde učení proběhne daleko rychleji, a jako vystup vrátí již serializovaného naučeného agenta. Tohoto agenta potom bude možno deserializovat a nahrát do grafického prostředí. S využitím v konzolové verzi bude také právě umožněn paralelní běh s použitím síťové komunikační knihovny Aeron s definovaným typem paralelní architektury a pak také s využitím vláken.

3.2.1 Umělá neuronová síť

Nejprve bylo nutné si určit strukturu samotné ANN [6], která je asi hlavní částí celého algoritmu. Z pohledu učení s využitím Q-learning algoritmu vychází nejlépe použití dopředné propagace, tedy Feed-Forward. V případě aktivační funkce použití funkce Sigmoid. Dále pak pro aktualizaci parametrů sítě a kompenzace chyby nalezené při učení využití metody Back-propagation.

Pro strukturu ANN byly hlavně definovány následující třídy:

- Neuron

  Jedná se o třídu popisující základní jednotku ANN, která obsahuje seznam vah, které do něj vstupují, inicializovanou aktivační funkci, hodnoty bias a lambda a také jejich chybové a derivační hodnoty. Nakonec pak vrací daný výstup generovaný aktivační funkcí ze sumy součinu vah a vstupů, který se použije pro další neurony jako vstup.

  - InputNeuron - Vstupní neuron

    Typ neuronu, jenž slouží čistě pro inicializaci vstupu sítě nacházející se pouze ve vstupní vrstvě. Obsahuje tedy pouze jenom jednu hodnotu, která se použije jako výstupní, bez toho aniž by prošla aktivační funkcí.

- IActivationFunction - Aktivační funkce

  Rozhraní aktivační funkce poskytující pouze metodu pro výpočet výstupu a metodu pro výpočet derivativního výstupu dané funkce. Slouží pro implementaci specifickou aktivační funkcí, která si tyto implementované metody definuje. Z hlediska nejlepší využitelnosti se v našem případě využívá aktivační funkce Sigmoid.

- Vrstva - Layer

  Abstraktní třída popisující ANN vrstvu, která obsahuje hlavně seznam svých neuronů. Dále pak obsahuje odkaz na předešlou a následující vrstvu, celkový výstup jako seznam výstupů všech neuronů a taky seznam všech chyb neuronů. Provádí pak inicializaci svých neuronů a

spouští nad neurony výpočty potřebné k dopředné propagaci a následně pak pro učení dle metody Back-Propagation.

Dělí se na jednotlivé odvozené typy:

- InputLayer - Vstupní vrstva

  Typ vrstvy, která se nachází na začátku ANN a slouží pouze pro inicializaci vstupu do sítě, kdy pak obsahuje seznam neuronů zahrnující pouze vstupní typ neuronu. Neodkazuje se na žádnou předcházející vrstvu a jako na následující vrstvu se odkazuje se na nějakou skrytou vrstvu. Dále neprovádí žádné výpočty a pouze poskytuje seznam výstupů svých neuronů, na nějž se odkazují následující vrstvy sítě.

- HiddenLayer - Skrytá vrstva

  Typ vrstvy, která se nachází uvnitř ANN, tedy mezi vstupní a výstupní vrstvou nebo mezi jinými skrytými vrstvami. Provádí pak nad svými neurony všechny výpočty a další potřebné záležitosti definované již základní abstraktní třídou pro vrstvu.

- OutputLayer - Výstupní vrstva

  Typ vrstvy, která se nachází na konci ANN a poskytuje tak celkový aproximovaný výstup ze sítě. Neodkazuje se tedy na žádnou následující vrstvu a pouze se odkazuje na nějakou předchozí skrytou vrstvu. Jinak provádí stejné výpočty, apod. definované základní abstraktní vrstvou.

- NeuralNet - samotná reprezentace ANN

  Třída, která zastává celou ANN, skládající se ze všech předešlých komponent. Obsahuje tedy vstupní vrstvu, seznam skrytých vrstev a výstupní vrstvu, které jsou na sebe postupně navázané. Provádí celkovou inicializaci ANN, kdy pak spouští jednotlivé inicializace v komponentách, jež obsahuje. Dále zahrnuje metodu pro vložení vstupu, po čemž spustí veškeré výpočty pro dopřednou propagaci, které hlavně deleguje na své komponenty, a pak z výstupní vrstvy získá požadovaný výstup. Následně provede v rámci učení aktualizaci svých parametrů dle metody Back-Propagation.

- Backpropagation

  Třída, která se stará o celkový průběh výpočtu realizovaného pomocí ANN. Pracuje s danou trénovací množinou a hlavně tedy s instancí třídy NeuralNet. Dále obsahuje data pro ukončovací kriteria a další potřebné konstanty a také metodu pro výpočet celkové chyby z výstupu sítě. Postupně tedy předává data z trénovací množiny instanci třídy NeuralNet, nad nimiž pak NeuralNet spustí potřebné výpočty pro průchod ANN a vrátí aproximovaný výstup. Poté se provede výpočet celkové chyby a ta se předá instanci NeuralNet pro zpětné šíření chyby v rámci ANN. Následně ještě zkontroluje ukončovací kriteria a pokud nejsou dosaženy, tak celý proces proběhne znovu s dalšími daty z trénovací množiny.

Průběh můžeme shrnout asi takto:

1. Inicializace parametrů a instance třídy NeuralNet
2. Trénink ANN dokud nejsou splněny ukončovací kriteria
   (a) Předání dat z trénovací množiny instanci NeuralNet
   (b) Spuštění metody Feed-Forward v rámci instance NeuralNet pro dopřednou propagaci
   (c) Získání výstupu, porovnání s očekávaným výstupem a výpočet celkové chyby
   (d) Spuštění učení pomocí metody Back-Propagation v rámci instance NeuralNet pro zpětné šíření chyby a adaptaci vah
3. Korektní ANN

3.2.2 Deep Q-learning

Pro samotné Deep Q-learning [9] učení je potřeba lehce upravit předchozí třídy ANN pro následné použití v DQN, která pracuje již tedy v rozhraní Agent-Environment dle zásad metody Reinforcement learning. Tento přístup pak také zahrnuje přidání nových tříd, dalších parametrů a zároveň několika technik určených k tomu, aby bylo učení více stabilní.

Hlavní nově přidané třídy DQN:

- State

  Třída popisující stav Agenta, ve kterém se v daném prostředí nachází, který je následně předávaný na vstup DQN. V případě vybraných prostředí Breakout a Pong zahrnuje 4 neordinální atributy reálných čísel:

  – xDif - rozdíl pozic X míčku a plošinky
  – yDif - rozdíl pozic Y míčku a plošinky
  – xBallVelocity - rychlost míčku X
  – yBallVelocity - rychlost míčku Y

- Action

  Třída popisující akci, kterou Agent v daném stavu koná a kterou DQN vrací jako výstup.

  Skládá se z:

  – inputType - název akce
  – value - hodnota akce

- Experience

  Třída popisující zkušenost Agenta, kterou si ukládá v každém časovém kroku do své tzv. Replay memory.

  Skládá se z:

  - state - aktualní stav s_t
  - action - akce a_t provedená ve stavu s_t
  - reward - odměna r_{t+1} získaná za provedenou akci a_t ve stavu s_t
  - newState - nový stav s_{t+1} získaný po provedení akce a_t
  - isDone - logická hodnota reprezentující zda je stav s_{t+1} terminální

- EpsilonGreedyPolicy

  Třída popisující výběr akce, kdy se jedná celkově o metodu, která z daného seznamu možných akcí vrátí akci dle proměnné *epsilon*, a to buď náhodnou akci nebo akci s největší hodnotou.

- AgentDQN

  Třída popisující AI Agenta, jenž koná akce v daném prostředí. Jeho "vědomí"tvoří jeho vlastní DQN, která slouží pro aproximaci nejvhodnějších akcí v daných stavech a pro následné učení dle odezvy prostředí na vykonanou akci v daném stavu. Pro výběr akce Agentovi slouží instance třídy EpsilonGreedyPolicy, která je použita po získání výstupu DQN. Agent také obsahuje vlastní Replay memory, což je seznam, kde si ukládá své získané zkušenosti v podobě instance třídy Experience. Dále pak obsahuje definovanou velikost dávky zkušeností pro učení a proměnou epsilon, na které závisí výběr akce, která se v průběhu snižuje. Z hlediska učení pak Agent operuje dle definovaného způsobu učení, jenž je udáváno výčtovým typem LearningType.

  Hlavní části:

  - replayMemory - seznam/trénovací množina uchovávaných zkušeností třídy Experience o dané velikosti
  - dqnBackpropagation - instance třídy DQNBackpropagation sloužící pro správu učení DQN
  - epsilonGreedyPolicy - instance třídy EpsilonGreedyPolicy sloužící pro výběr akcí
  - learningType - výčtový typ způsobu učení Agenta
  - epsilon - proměnná sloužící pro balancovaní výběru akcí
  - batchSize - velikost dávky náhodně vybírané z replayMemory

Třídy odvozené nebo upravené z předchozích ANN tříd:

- DQNOutpuLayer - Výstupní vrstva DQN

  Z hlediska vrstev stačí upravit pouze výstupní vrstvu OutputLayer, jelikož v případě DQN se na výstupu nacházejí přímo dané akce určené pro Agenta. Proto tato vrstva obsahuje seznam akcí třídy Action, které jsou namapovány z neuronů po provedení výpočtu. Tento seznam akcí je pak využit pro výběr nejvhodnější akce.

- DQNNeuralNet - samotná reprezentace DQN

  Třída odvozená z třídy NeuralNet pro potřeby DQN. Obsahuje tedy hlavně jinou výstupní vrsvu typu DQNOutputLayer a podle ní přizpůsobené metody pro inicializaci DQN, dopřednou propagaci a zpětné šíření chyby. Na vstupu tedy přijímá daný stav třídy State a po průchodu DQN vrací seznam akcí třídy Action.

- DQNBackpropagation

  Třída popisující specificky celkový průběh výpočtu pomocí DQN. Je tvořena a pracuje trochu jinak než předchozí třída Backpropagation pro ANN. Nepracuje již tedy s nějakou trénovací množinou, ale pracuje nejprve s daným stavem, který je předán od Agenta pro získání další akce. Tento stav je poté předán hlavní DQN - tzv. Policy network, ze které jsou pak vráceny aproximované hodnoty všech akcí, které si Agent zpracuje dle své politiky výběru akce.

  Dále od Agenta získává dávky zkušeností třídy Experiences. Stavy z jednotlivých zkušeností z těchto dávek jsou poté předávány na vstup DQN, jenž je realizovaná třídou DQNNeuralNet. Dle poznatků získaných z předchozí teoretické analýzy je právě stav s_t předán Policy network a stav s_{t+1} je předán podle způsobu učení buď také Policy network nebo právě Target network, což je kopie Policy network, která se vždy periodicky opět zkopíruje a pomáhá k stabilnějšímu učení dle tedy způsobu učení. Po získání těchto dvou výstupů, je pak vypočítána loss funkce, podle které je aktualizována Policy network pomocí metody Back-Propagation.

  Pak je v této třídě implementována také volitelná paralelizace pomocí vláken v podobě tzv. ThreadPoolu. Dle definovaného počtu vláken se pak dávka zkušeností rozdělí na díly a ty jsou pak předány jednotlivým vláknům na paralelní zpracování. Vlákno je reprezentováno třídou DQNWorker. Tato vlákna jsou následně vložena do seznamu, se kterým ThreadPool pracuje, jenž vždy všechny vlákna najednou spustí a pak počká na dokončení práce všech těchto vláken pro pokračování dalšího běhu programu. Vláknu je pak vždy předána aktuální Policy a Target network a díl dané dávky, který je pak zpracován jeho vlastní instancí třídy DQNBackpropagation. Po zpracování daného dílu jsou potřebné parametry Policy network (váhy, bias, lambda) překopírovány do sdílené paměti vláken a následně je tato paměť zprůměrována podle definovaného počtu vláken. Těmito parametry jsou poté nahrazeny parametry hlavní Policy network. Tento způsob však nevyhovuje všem dostupným způsobům uvedeným dále.

3.2.2.1 Způsoby učení

Pro samotné učení typu Deep Q-learning, pak byly navrženy tři následující způsoby učení, které mají potenciál pro úspěšné zvládnutí nebo pokoření zvolených testovacích prostředí.

3.2.2.1.1 STEPS - časové kroky Jedná se, dle předchozí rešerše, o asi nejpoužívanější a nejefektivnější typ učení. V rámci nějž se používá rozdělení učení na Policy a Target network. Jak už bylo

nastíněno dříve, tak Policy network slouží jako hlavní síť, která je trénována a poskytuje Agentovy aproximovaný výstup možných akcí a Target network pak reprezentuje periodicky se obnovující kopii Policy network, která slouží při výpočtu Q-learning loss funkce k lepší stabilitě učení pomocí metody Back-propagation.

Toto učení tedy probíhá po dosažení dostatečné kapacity zkušeností ukládaných postupně v každém časovém kroku do Agentovy Replay memory, kdy se pak v každém časovém kroku z Replay memory vybere náhodná dávka zkušeností tzv. minibatch o definované velikosti a ta se následně předá DQN na zpracování.

3.2.2.1.2 EPISODES - epizody V tomto případě jde o určitou vlastní invenci, kdy se pracuje jenom s Policy network. Trénování DQN však již právě neprobíhá v každém časovém kroku s náhodně vybíranou dávkou z Replay Memory, ale vždy se zpracují všechny zkušenosti získané za jednu epizodu daného testovacího prostředí. Epizodou se myslí běh prostředí, ve kterém se v každém časovém kroku zapisují zkušenosti do Replay memory do té doby, dokud se agent nedostane do terminálního stavu, kdy prostředí pak musí být resetováno. Po resetu prostředí se vezme celá Replay memory s nastřádanými zkušenostmi za jednu epizodu a předá se DQN na zpracování. Poté se Replay memory vymaže a zase probíhá postupné přidávání zkušenosti v další epizodě prostředí.

Výhodou je nějaká posloupnost zpracovávány zkušeností, ale zase nevýhodou je postupně zvyšující se dávka zkušeností, která se při dosažení terminálního stavu musí najednou zpracovat, což má dopad na celkový výkon aplikace. Tento problém lze poté řešit právě paralelizací pomocí vláken.

3.2.2.1.3 CYCLES - cykly Tento způsob je podobný předchozímu epizodickému způsobu, kdy se jedná rovněž o určitou vlastní invenci a pracuje se také jenom s Policy network. Trénování DQN tedy probíhá podobným způsobem jako je tomu u epizodického způsobu, kdy se zkušenosti opět ukládají v každém časovém kroku, a pak se v určitém stavu prostředí vezme celá Replay memory, předá se DQN na zpracování a poté je vymazána. Tento stav pro aktualizaci nastává právě po daných cyklech, jenž jsou v rámci zvolených testovacích prostředí definovány, že nastanou když:

- Agent se dostane do terminálního stavu a prostředí musí být resetováno
- Agent úspěšně odpálí míček

V tomto případě je zachována jakási posloupnost zpracovávaných zkušeností a zároveň zpracování probíhá v kratších časových úsecích s menšími velikostmi dávek zkušeností, takže následný trénink nemá takový dopad na výkon aplikace. I když při čím dál větší úspěšnosti Agenta se ale tyto dávky zvětšují, a je proto také dobré využít paralelizace pomocí vláken.

3.3 Návrh prostředí

Jak bylo již nastíněno, tak pro běh Reinforcement learning algoritmu je kromě Agenta nutné poskytnout určité stochastické prostředí, ve kterém Agent může určitým způsobem operovat a rozhodovat se, čímž mění stav prostředí a dostává od něj patřičnou odezvu.

V tomto případě byly pro implementaci zvoleny dva typy prostředí z kolekce arkádových her od společnosti Atari - Breakout a Pong. Tyto prostředí byly vybrány proto, že jsou dostatečně netriviální a umožňují činnost daného AI Agenta ovládajícího ve hře danou entitu. Ten může volbou svých akcí měnit stav hry a z hlediska své úspěšnosti je pak hodnocen podobně jako normální lidský hráč. Pro hodnocení agenta za jeho zvolené akce je nutné v daném prostředí definovat, jakým způsobem bude při určitých událostech v jednotlivých časových krocích vracet Agentovi odměny o dané velikosti. Tyto akce, které může Agent volit, musí být prostředím jasně definovány a poskytnuty. Při inicializaci Agenta jsou pak tyto akce v rámci Agentovi DQN realizovány výstupní vrstvou neuronů, kdy jeden neuron reprezentuje jednu akci.

Následuje popis zvolených prostředí, jejich poskytovaných akcí a odměn.

3.3.1 Breakout

V tomto prostředí Agent ovládá plošinku ve spodní části herní plochy pohybující se horizontálně. Jeho úkolem je odrazit v prostředí se pohybující míček, a to nejlépe ještě takovým způsobem, aby pak následně míček rozbil co nejvíce cihel nacházejících se v horní části herní plochy. Tento míček lze také odrazit pod různým úhlem při dopadu na určitou část plošinky. Při neúspěšné odrazení míčku se hra resetuje.

Definované typy akcí v prostředí:

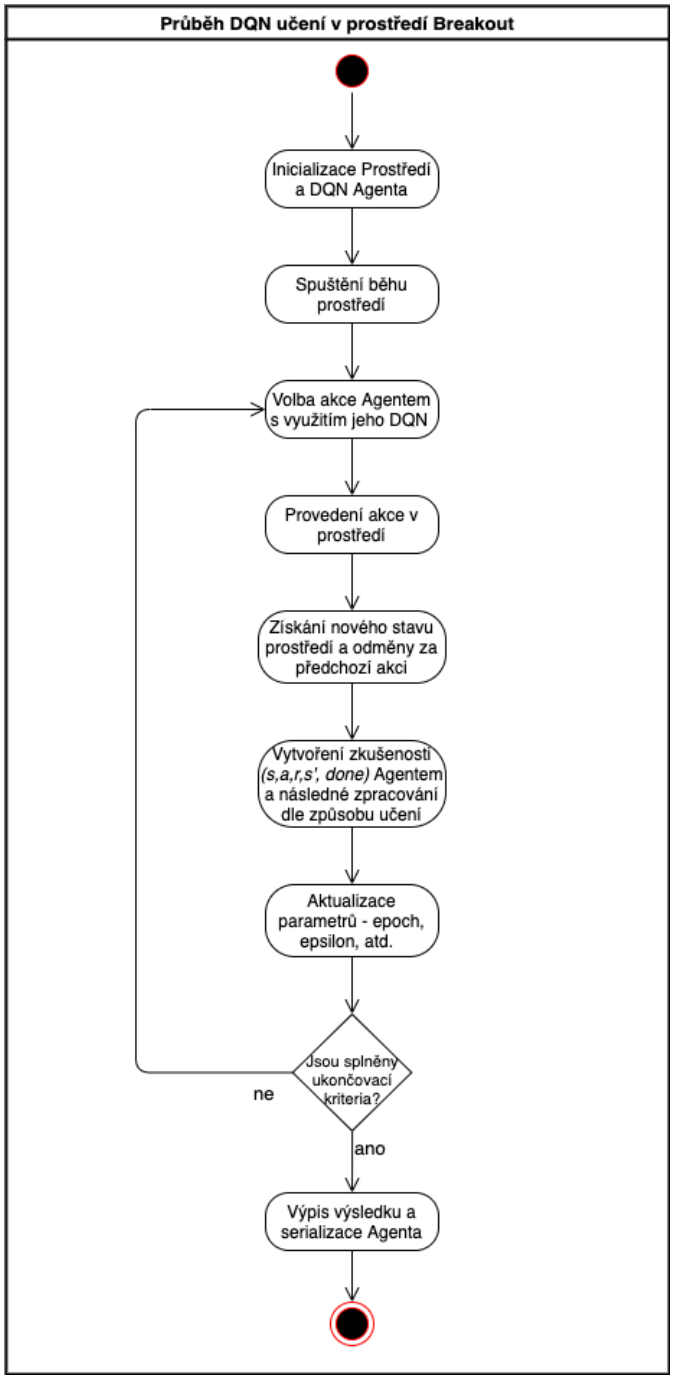
- NONE - žádný pohyb
- LEFT - pohyb plošinky doleva
- RIGHT - pohyb plošinky doprava

Výchozí definované nominální odměny v prostředí:

- NORMAL_VALUE = 1.0 - základní standartní odměna při nevýznamné událostí
- CORRECT_MOVE_VALUE = 5.0 - pohyb směrem k místu dopadu míčku
- WRONG_MOVE_VALUE = -5.0 - pohyb směrem od místa dopadu míčku
- SUCCESS_BALL_HIT_VALUE = 10.0 - úspěšné odražení míčku
- FAILED_BALL_HIT_VALUE = -10.0 - neúspěšné odražení míčku nebo-li terminální stav
- BRICK_HIT_VALUE = 10.0 - úspěšný zásah cihly míčkem

3.3.1.1 Průběh učení v kontextu prostředí

Průběh Deep Q-learning učení v testovacím prostředí Breakout můžeme vidět na následujícím aktivitním diagramu.



Obrázek 3.2: Průběh DQN učení v testovacím prostředí Breakout

3.3.2 Pong

V tomto prostředí operují přímo dva Agenti, jenž ovládají každý svou plošinku, které jsou umístěny na levé a pravé straně herní plochy pohybující se vertikálně. Jejich úkolem je odrazit v prostředí se pohybující míček ještě nejlépe takovým způsobem, aby ho jejich protihráč nestihl odrazit, a tak získali bod do svého skóre. Míček jde pak odrazit pod různým úhlem dle dopadu na určitou část plošinky. V případě, že daný Agent nestihne odpálit míček, tak se hra resetuje, protihráči se přičtou body a tento Agent má podání.

Definované typy akcí v prostředí:

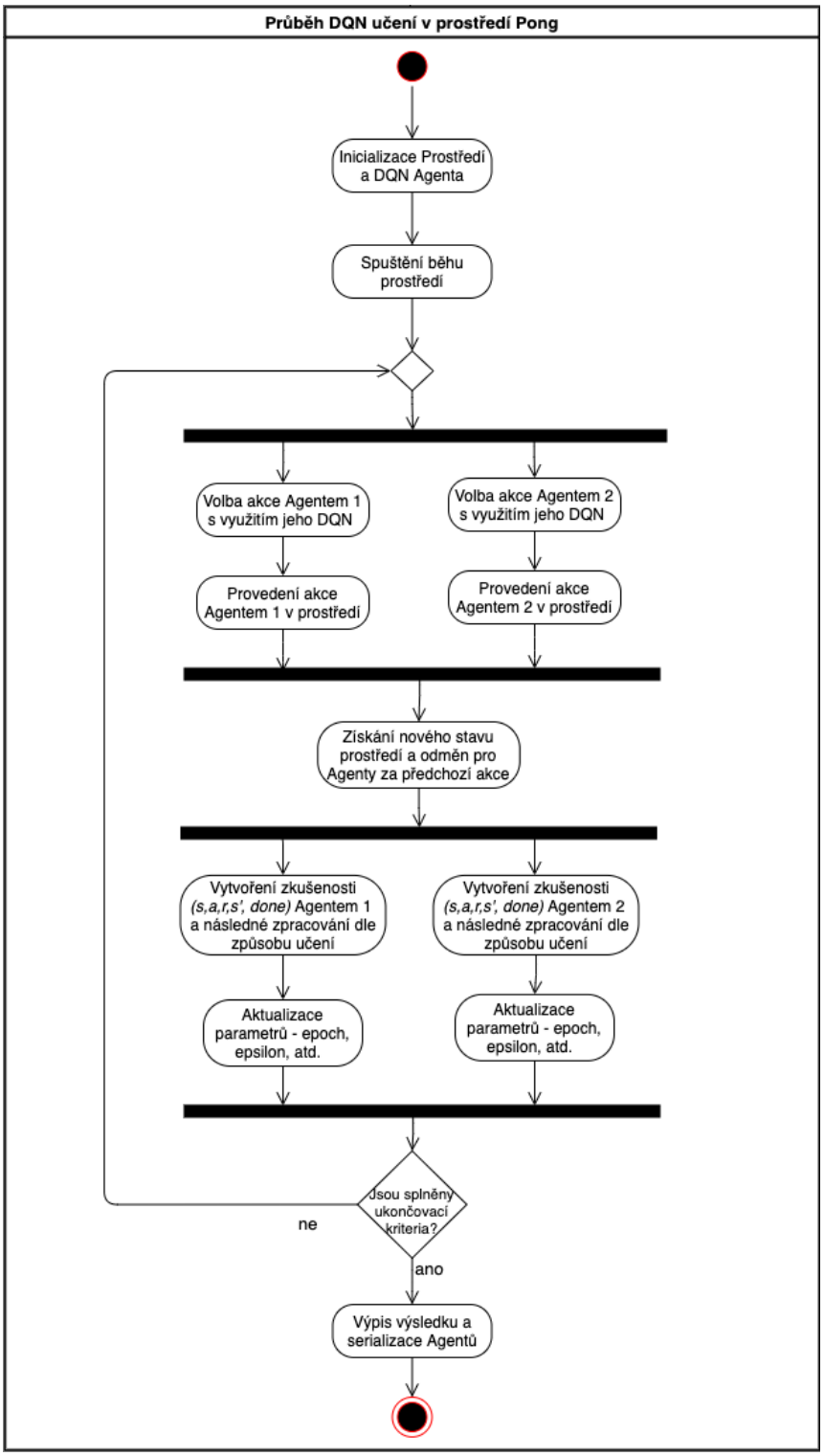
- NONE_ONE - žádný pohyb prvního hráče
- UP - pohyb plošinky prvního hráče nahoru
- DOWN - pohyb plošinky prvního hráče dolů
- NONE_TWO - žádný pohyb druhého hráče
- W - pohyb plošinky druhého hráče nahoru
- S - pohyb plošinky druhého hráče dolů

Výchozí definované nominální odměny v prostředí:

- NORMAL_VALUE = 1.0 - základní standartní odměna při nevýznamné událostí
- CORRECT_MOVE_VALUE = 5.0 - pohyb směrem k místu dopadu míčku
- WRONG_MOVE_VALUE = -5.0 - pohyb směrem od místa dopadu míčku
- SUCCESS_BALL_HIT_VALUE = 10.0 - úspěšné odražení míčku
- FAILED_BALL_HIT_VALUE = -10.0 - neúspěšné odražení míčku nebo-li terminální stav
- SUCCESSFUL_POINT_VALUE = 20.0 - úspěšné získání bodu při chybě protihráče

3.3.2.1 Průběh učení v kontextu prostředí

Průběh Deep Q-learning učení v testovacím prostředí Pong můžeme vidět na následujícím aktivitním diagramu.



Obrázek 3.3: Průběh DQN učení v testovacím prostředí Pong

3.4 Návrh paralelních komunikačních modelů

Pro účely zlepšení rychlosti a demonstrace distribuovaného paralelního výpočtu v kontextu DQN je potřeba zvážit dostupné přístupy k tomuto problému. Tyto přístupy již byly zmíněny dříve a jsou jimi modelový a datový paralelismus. V souvislosti s těmito přístupy byl pro účely této práce tedy zvolen způsob paralelismu postavený na principu datového paralelismu. A to hlavně kvůli vhodnějšímu modelu pro náš problém a také kvůli komplexnosti implementace modelového paralelismu.

Dále přímo rychlost Deep Q-learning učení je ještě oproti klasickém učení umělých neuronových sítí limitována hlavně dvěma následujícími faktory:

- komplexnosti vzoru - např. počet přechodů potřebných k osvojení uspokojivé rozhodovací politiky
- online interakcí s prostředím

S tímto uvážením je zde pak přístup datového paralelismu reprezentován návrhem daných paralelních modelů sloužícím pro dané navržené způsoby učení popsaných dříve. Jedná se tedy o modely Master-Slave a Gorilla DQN.

3.4.1 Paralelní model Master-Slave

Model asymetrické komunikace Master-Slave je v informatice celkem využívaný model v různých odvětvích. V tomto případě návrhu paralelismu pak celkem vystihuje princip datového paralelismu. Jedná se tedy o ansynchronní centralizovaný způsob komunikace mezi uzly paralelizace, kdy hlavním prvkem je tzv. Master, který udržuje vždy hlavní aktuální kopii DNN a také dataset pro potřeby učení. Dalším prvkem je tzv. Slave, který reprezentuje prvek provádějící příjem a zpracování dat, které dostane o Mastera. Master vždy pak zasílá na adresu Slave uzlu aktuální kopii DNN a díl daného datasetu. Slave pak po zpracování distribuuje na adresu Master uzlu jeho DNN parametry nebo případně jiná data např. gradienty. Je tedy vidět, že Master i Slave jsou oba takto příjemci i odesílatelé. Dále pak Master vždy po získání DNN dat od uzlu Slave musí provést aktualizaci své hlavní DNN, kterou drží. Tato DNN je pak výstupem celého distribuovaného paralelního výpočtu.

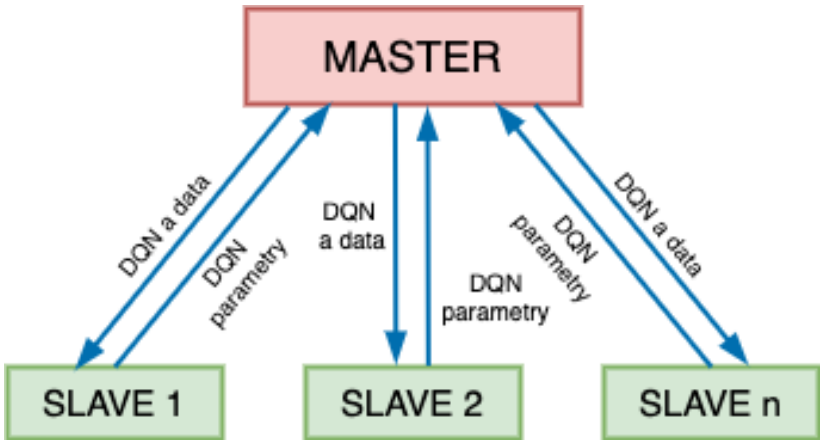
Komunikace v tomto případě není až tak složitá, ale má velké nároky na rychlost a spolehlivost přenosu dat. Řízení komunikace a celkové řešení je realizováno pomocí knihovny Aeron [19], která se toho v tomto kontextu snaží co nejlépe dosáhnout navzdory tomu, že používá nespolehlivý způsob komunikace přes protokol UDP.

Samotné řízení komunikace dle celkového průběhu učení v rámci interakce agenta s testovacím prostředím je nutno pak také kvůli dostatečné přehlednosti určitým způsobem specifikovat. Pro toto řešení je potřeba vytvořit systém daných stavů, které budou popisovat, v jaké situaci se daný uzel zrovna nachází, a podle toho přizpůsobovat vykonávání programu.

Všechny možné stavy jsou zachyceny v následujícím výčtu:

- Stav NOT_INIT - uzel není inicializován
- Stav WAITING_FOR_DATA - uzel je připraven na příjem dat
- Stav SEND - na daný uzel byla odeslána data
- Stav TERMINATED - činnost uzlu byla ukončena

Z hlediska použití v případě této implementace Master uzel obsahuje prostředí běžící na vlastním vlákně a hlavního Agenta s DQN, který provádí akce v tomto prostředí. Potom vždy když prostředí dojde do terminálního stavu, tak Master vezme celou Replay memory a rozdělí ji podle počtu dostupných Slave uzlů. Poté po úspěšném odeslání všech dílů Replay memory a aktuální DQN hlavního Agenta, spouští další běh prostředí, dokud se opět nedostane do terminálního stavu, kdy se poté celý proces opakuje. Současně také stále od Slave uzlů očekává buď žádost o data nebo již nové DQN parametry po zpracovaní, které potom aktualizuje svému Agentovi. Po dosažení ukončovacích kriterií je pak výstup Master uzlu již naučený Agent se svou DQN.



Obrázek 3.4: Paralelní komunikační model Master-Slave

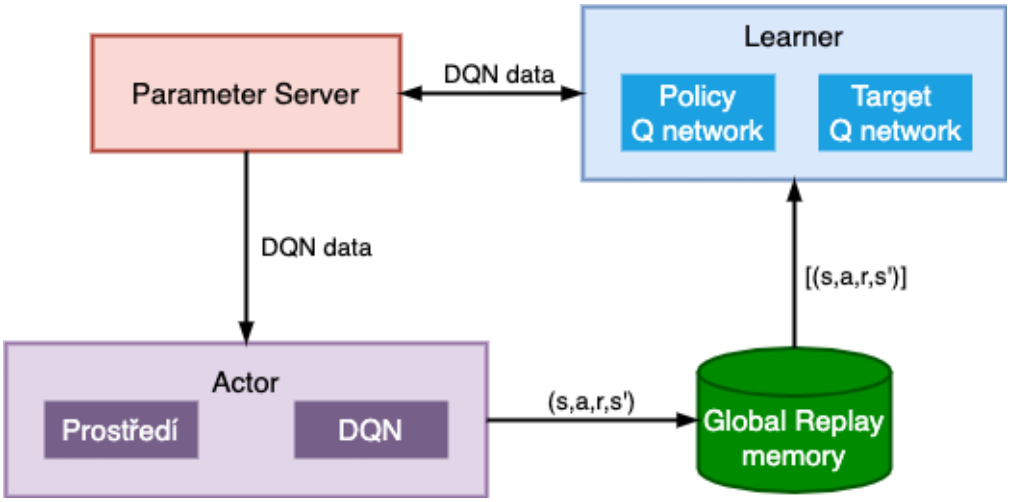
3.4.2 Paralelní model Gorilla DQN

Jedná se o model, který je také určitým způsobem postaven na datovém paralelismu a je na rozdíl od modelu Master-Slave určen přímo pro distribuované paralelní řešení Deep Q-learning učení.

V modelu Gorilla DQN více agentů tzv. *Actors*, interagují s prostředím a shromažďují zkušenosti. Každý Actor má nezávislou kopii prostředí, takže může shromáždit N krát více vzorků za sekundu, pokud jich existuje N počet. Tyto zkušenosti odesílají do tzv. *Global replay memory*, jenž shromažďuje všechny nasbírané zkušenosti od všech agentů typu Actor. Pak se zde nachází další typ agentů tzv. *Learners*, kteří neinteragují s prostředím, ale provádí výpočet pomocí DQN s využitím Policy a Target network. K získaní dat pro výpočet se musí vždy zaslat požadavek na Global replay

memory, která jim obratem odešle náhodnou dávku dat o definované velikosti. Po zpracování dat pak každý Learner zasílá parametry své DQN na tzv. *Parameter Server*, jenž udržuje hlavní aktuální DQN, kterou vždy po získání dat od agenta typu Learner aktualizuje. Tuto DQN pak vždy po aktualizaci nebo periodicky rozešle všem agentům typu Actor i typu Learner, kteří si jí nahradí svou stávající DQN. Takto proces probíhá až do splnění ukončovacích kriterií.

Komunikace je řízena stejně jako u modelu Master-Slave, tedy pomocí knihovny Aeron [19], která se stará o zajištění rychlého a spolehlivého přenosu dat. Dále pak samotné řízení komunikace z hlediska učení musí být opět specifikováno systémem stavů, které popisují situaci, ve které se daný uzel nachází. Výčet těchto stavů uzlů je shodný s výčtem stavů modelu Master-Slave.



Obrázek 3.5: Paralelní komunikační model Gorilla DQN

Kapitola 4

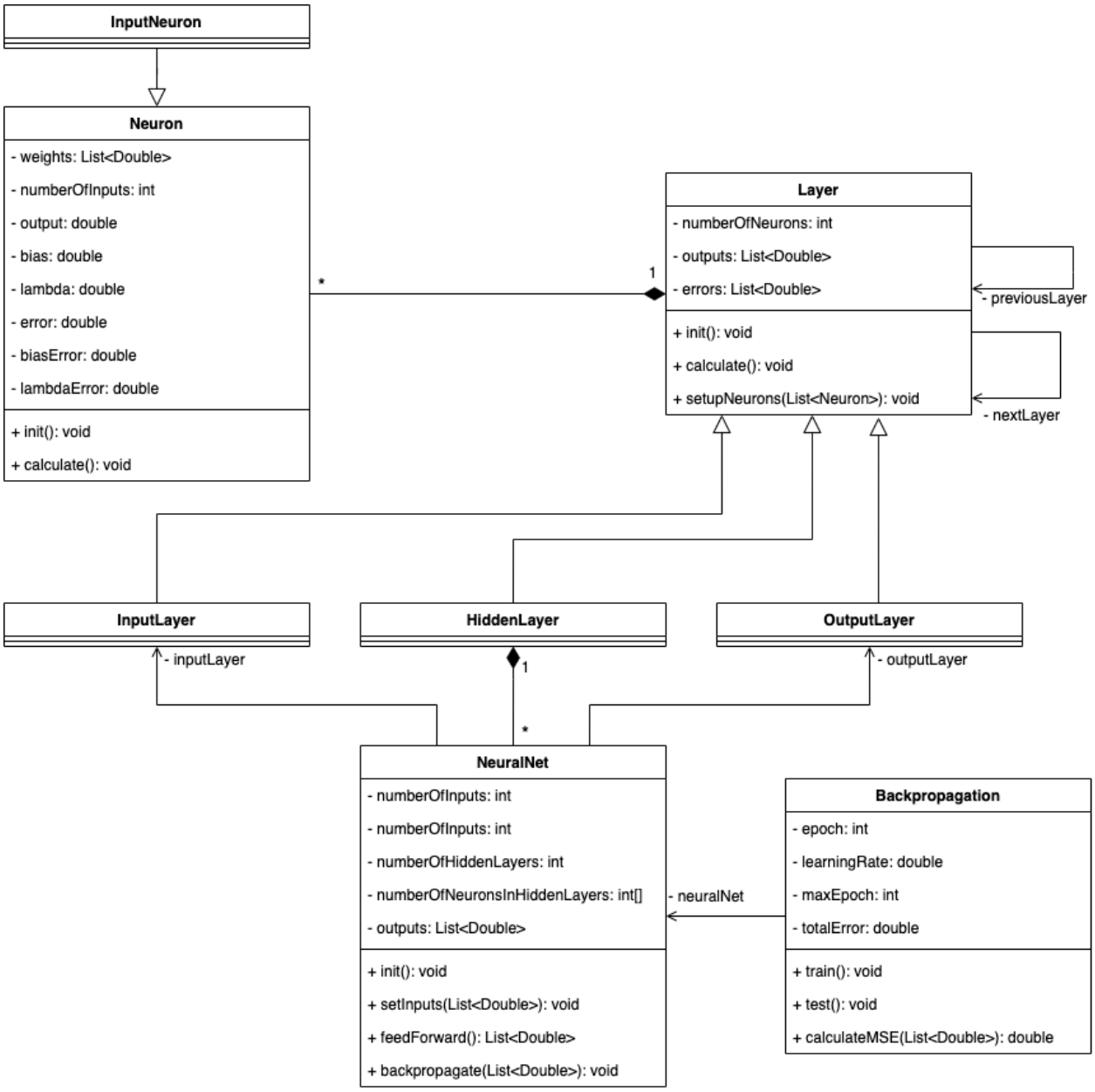
Implementace

V této kapitole je postupně popsána samotná implementace dle předchozího návrhu. Jsou zde tedy zahrnuty implementační detaily jednotlivých částí projektu a jejich propojení. Také jsou zde uvedeny určité potíže, které v průběhu implementace nastaly.

4.1 Implementace umělé neuronové sítě a Deep Q-learning algoritmu

Nejprve byla nutná implementace samotné Feed-Forward ANN, tedy umělé neuronové sítě s dopřednou propagací a metodou Back-Propagation pro učení z hlediska zpětného šíření chyby. Tato realizace probíhala dle předchozího návrhu, kdy byly postupně implementovány dané navržené třídy. Tato ANN byla pak ještě celkově realizovaná jako hluboká/vícevrstvá, takže tedy typu DNN. [6]

Celková výsledná realizace DNN je vidět na UML třídním diagramu na obrázku 4.1.

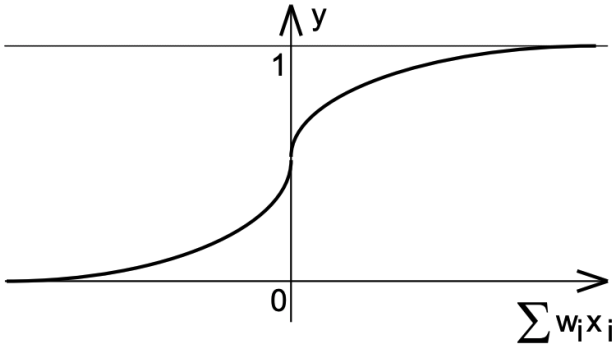


Obrázek 4.1: Třídní diagram DNN [19]

Na počátku byla DNN implementována hlavně pro adaptací synaptických vah mezi neurony a neurony měly stejnou aktivační funkci, tedy stejnou strmost sigmoidu. Jednalo se tak o tzv. homogenní typ DNN, která pracuje po celou dobu výpočtu v rámci neuronů s prahovými hodnotami *lambda* a *bias* jako s neměnnými konstantami a tím strmost sigmoidu zůstává stejná.

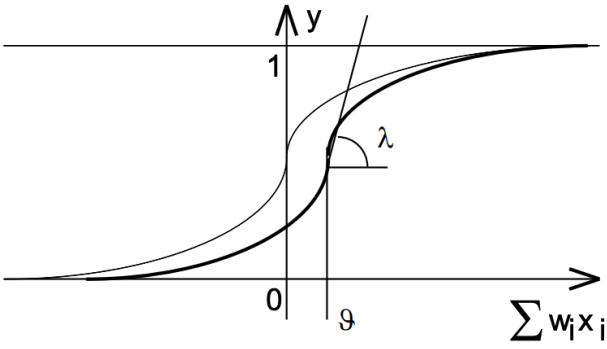
Proto z hlediska zlepšení efektivity a citlivosti učení DNN bylo třeba u neuronů podrobit adaptaci i prahové hodnoty a strmosti sigmoidů. Po této úpravě tak můžeme získat tzv. heterogenní DNN, kde pak každý neuron může mít svou vlastní aktivační dynamiku a díky tomu se pak zvyšuje

schopnost sítě konvergovat k naučenému stavu. Tato metoda se pak nazývá tzv. parametrická Back-Propagation. Podstatou této metody je pak definovaná chybová funkce závislá nejen na vektoru synaptických vah, ale i právě na vektoru strmostí sigmoidů a prahů. Po této úpravě pak DNN tedy dosahovala lepších výsledků. Potom pro počáteční testování DNN implementace byl použit XOR problém.



$$y = \frac{1}{1 + e^{-x}}$$

Obrázek 4.2: Aktivační funkce Sigmoid [20]

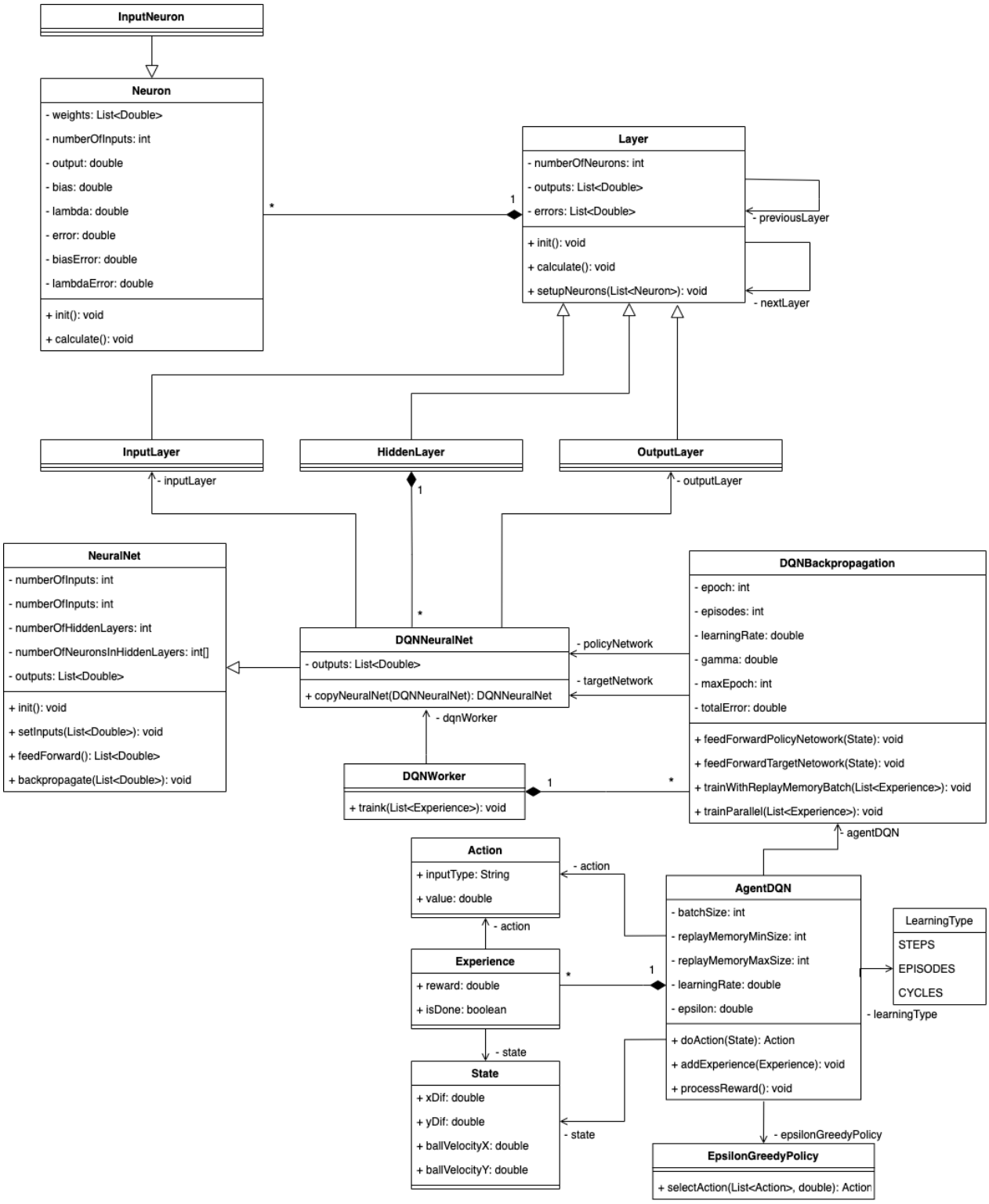


$$y = \frac{1}{1 + e^{-\lambda(x-\delta)}}$$

Obrázek 4.3: Parametrizovaná aktivační funkce Sigmoid [20]

Dále po úspěšné implementaci a otestování DNN bylo možné začít s propojením pro účely Deep Q-learning účení, tedy s implementací DQN postavené na základech DNN. Proto byly vytvořeny odvozené třídy od DNN tříd a potřebné nové třídy přímo pro potřeby DQN dle předchozího návrhu. Ještě z hlediska čistě implementačních detailů byl pro třídu AgentDQN, jelikož se jedná o složitější třídu s případným přidáváním další funkcionality v průběhu implementace, použit návrhový vzor *Stavitel/Builder*, jenž tuto konstrukci zobecňuje a snižuje tak i duplicitu kódu. V třídě AgentDQN je pak také realizováno celkové řízení s ohledem na jeden ze zvolených způsobů učení - STEPS, EPISODES, CYCLES, popsaných v návrhu.

Výsledná realizace DQN je vidět na UML třídním diagramu na obrázku 4.4.



Obrázek 4.4: Třídní diagram DQN [19]

4.2 Implementace prostředí

Další bodem implementace bylo prostředí pro testování DQN. Jak už bylo zmíněno dříve, tak toto prostředí bylo vyvíjeno s využitím grafické knihovny JavaFX [17]. Implementovány byly pak dvě zvolené testovací prostředí arkádových her Atari - Breakout a Pong [12].

Ohledně těchto prostředí bylo nutné vytvořit hlavně samotný panel pro běh prostředí reprezentovaný u těchto prostředí třídou `Game`, která implementuje abstraktní třídu `GameType`, poskytující hlavní jednotné rozhraní pro práci. Dále celkové hlavní rozhraní aplikace, reprezentované třídou `[Breakout|Pong]Main`, která kombinuje již zmíněný panel pro běh prostředí také s bočním panelem zobrazujícím aktuální informace o prostředí a také částečnou konfiguraci za běhu prostředí. Na závěr bylo celkově potřeba tyto prostředí poskytnout v rámci jednoho spouštěcího rozhraní pro lepší uživatelský komfort práce jako se samostatnou aplikací.

4.2.1 Entity prostředí

Obě herní testovací prostředí jsou si podobné a tak obsahují a využívají stejné typy entit nebo modelů. Tyto entity jsou odvozeny od abstraktní třídy `Entity`, která popisuje hlavní vlastnosti dané entity jako výšku, šířku, pozici a rychlosti a dále pak chování týkající se aktualizace těchto parametrů, apod. Následující vyjmenované odvozené třídy si pak případně přidávají další své vlastní vlastnosti a chování.

- Ball - třída reprezentující míček pohybující se v prostředí
- Paddle - třída reprezentující plošinku/pálku, která odráží míček a jenž ovládá AI agent
- BallMovement - výčtový typ určující, zda se míček pohybuje nahoru nebo dolů, sloužící hlavně pro potřeby AI agenta
- Brick - třída reprezentující cihlu v prostředí Breakout, která je po kolizi s míčkem zničena

4.2.2 Ovládání prostředí

Z technického hlediska obou herních testovacích prostředí byly nejdříve implementovány samotné hry ovládané uživatelskými vstupy z klávesnice. Pro zachycení těchto vstupů bylo potřeba implementovat globální ovladač zachycující vstupy hlavní JavaFX scény, které pak využije samotná hra kdekoliv ve své herní smyčce. Pro nutnost využití tohoto ovladače globálně byl využit návrhový vzor *Jedináček/Singleton*, který právě poskytuje možnost existence pouze jedné instance určité třídy a k ní pak také globální přístupový bod. Tento ovladač pak poskytuje hlavně seznam zachycených vstupů definovaných v předchozím návrhu.

Tento ovladač byl poté použit také přímo pro samotnou AI Agenta reprezentovanou třídou `AgentDQN`. Tato třída obsahuje metodu `makeMove`, která předá aktuální stav prostředí na vstup

DQN. DQN pak aproximuje hodnoty možných akcí, ze kterých je následně vybrána akce dle politiky Epsilon-Greedy policy. Daný vstup obsažený ve vybrané akci, je pak vložen do ovladače, pro následné provedení v herní smyčce.

4.2.3 Herní smyčka prostředí

Herní smyčka prostředí představuje "hlavní centrum řízení"celého běhu prostředí i algoritmu, jak je nastíněno v diagramu aktivit v předchozím návrhu. V této smyčce se tedy nachází samotné volání metod pro aktualizaci prostředí, volání metod pro výpočet odměny prostředí, metod pro výpočet různých pomocných údajů a aktualizaci UI statistik a také hlavně volání metod pro zapojení Deep Q-learning učení pomocí třídy `AgentDQN`. Od instance této třídy přijímá pomocí ovladače prostředí dané vstupy z vybraných akcí, předává mu odměny a podklady pro vytvoření zkušeností a spouští zpracování zkušeností dle definovaného způsobu učení agenta.

K realizaci byl použit `AnimationTimer`, který volá metodu `handle` v každém snímku za sekundu a ta poskytuje časové razítko aktuálního snímku udávané v nanosekundách, které se použilo pro aktualizaci a přepočet pozice míčku kvůli plynulosti samotného běhu. Později se však k využití prostředí jako konzolové verze musel tento způsob aktualizace pozměnit, jak je popsáno dále.

4.2.4 Spouštění prostředí

Jedním z hlavních počátečních problémů bylo samotné propojení AI agenta s prostředím. Dále jejich inicializace a spuštění, které by bylo určitým způsobem nezávislé na daném typu prostředí. Také bylo právě z hlediska celkového spuštění nutné první definovat a vytvořit instanci třídy AgentDQN s definici jeho DQN a poté spustit samotné prostředí, kterému se tento agent předá při inicializaci a až pak se spustí celkový běh.

Pro tyto účely byla poté definována třída `GameLauncher`, která rozšiřuje JavaFX aplikaci. Této třídě musí být, ještě hlavně kvůli politice spouštění JavaFX aplikací, nejdříve staticky definovány její proměnné. Tyto proměnné zahrnují hru, která má být spuštěna, což je definováno výčtovým typem `AvailableGames`, který definuje typy dostupných prostředí a dále pak seznam AI agentů třídy `AgentDQN`. Další proměnou je abstraktní třída `GameType`, kterou tyto jednotlivé prostředí implementují, díky čemuž jsou jim pak předáni daní AI agenti a také si musí definovat metodu pro start běhu prostředí kvůli jednotnému způsobu spuštění. Celý proces se po inicializaci spustí zavoláním metody `launch` na třídu `GameLauncher`, čímž se spustí celá JavaFX aplikace nebo respektive vlastním způsobem přepsaná JavaFX metoda `start`, kde se dle dříve nastavené proměnné výčtového typu `AvailableGames` inicializuje přímo dané prostředí s poskytnutými agenty a následně se spustí celý běh aplikace v rámci prostředí.

```
public class GameLauncher extends Application {
    public static AvailableGames gameToLaunch = null;
    public static GameType runningGame = null;
    public static List<AgentDQN> agents = null;

    public GameLauncher() {}

    public static void launch() { Application.launch(GameLauncher.class); }

    @Override
    public void start(Stage primaryStage) throws Exception {
        if(gameToLaunch == null || agents == null){
            StringBuilder stringBuilder = new StringBuilder("Game setup is missing:
                 ");
            if(gameToLaunch == null) {
                stringBuilder.append("Game type to launch,");
            }
            if(agents == null){
                stringBuilder.append("AI Agent are not specified.");
            }
            throw new InvalidGameSetupException(stringBuilder.toString());
        }

        switch (gameToLaunch){
            case PONG:
                PongMain pongMain = new PongMain();
                runningGame = pongMain.getGame();
                runningGame.setAgents(agents);
                pongMain.launchPong();
                break;
            case BREAKOUT:
                BreakoutMain breakoutMain = new BreakoutMain();
                runningGame = breakoutMain.getGame();
                runningGame.setAgentDQN(agents.get(0));
                breakoutMain.launchBreakout();
                break;
        }
```

```
    }
}
```

Listing 4.1: Třída GameLauncher

4.2.5 Konzolová verze prostředí

Verze testovacího herního prostředí s grafickým rozhraním sloužila tedy pro prvotní experimenty, ale potom v závislosti zejména na požadavek distribuovaného spouštění na výpočetních uzlech, nebyla pro tyto účely použitelná. Dále i z potřeby rychlejšího a obecně paralelního učení také nebyla vhodná.

Proto byla vytvořena kopie hlavní třídy prostředí `Game` reprezentovaná třídou `ConsoleGame` pro každé prostředí. V ní byly všechny potřebné třídy pro běh prostředí uzpůsobeny, tak aby nebyly závislé na JavaFX vlákně. Hlavním problémem byl právě již dříve zmíněný `AnimationTimer`, který není možné spustit mimo JavaFX vlákno. V němž tedy běžela hlavní smyčka programu a který také daným entitám v průběhu předával jeho aktuální časové razítko pro jejich aktualizaci. Proto bylo nutné přijít s novým přístupem, který poté zahrnoval to, že hlavní smyčka poběží v cyklu while a hodnota časového razítka bude konstantní, což tedy problém vyřešilo. Ještě pak z hlediska následného spouštění naučeného serializovaného agenta v grafickém rozhraní, bylo nutné upravit aktualizaci entity v hlavní herní smyčce, kterou reprezentuje `AnimationTimer`, také dle konstantního časového razítka.

Tato verze tedy nepotřebuje být spouštěna pomocí třídy `GameLauncher`. Dále po dosažení ukončovacích kritérii vypíše na výstup do konzole údaje o shrnutí výpočtu a serializuje do souboru třídu AI agenta `AgentDQN`, která poté může by deserializována pro případnou demonstraci, apod.

4.3 Implementace paralelizace

Pro účely paralelizace Deep Q-learning učení [10] byly navrženy dle předchozího návrhu tři způsoby. Prvním je paralelizace pomocí vláken určená pro lokální konzolovou verzi, další dva jsou paralelní modely určené hlavně pro distribuovanou síťovou paralelizaci na výpočetních uzlech s využitím knihovny Aeron [19].

4.3.1 Vlákna

Tento paralelní způsob byl vytvořen jako první a slouží pro paralelizaci hlavně v rámci lokální konzolové verze. Tento způsob je implementován třídou `DQNWorker`, která popisuje definované vlákno, které umožňuje zpracování přidělených dat dle metody Back-Propagation. Ke svému běhu potřebuje při inicializaci přidělit dávku zkušeností třídy `Experience` a také aktuální kopii DQN, která

je reprezentována třídou `DQNNeuralNet` pomocí, které může provést výpočet na danou dávkou zkušeností. Tato třída je pak využívána třídou `DQNBackpropagation`, která vždy při požadavku na zpracování dávky zkušeností tuto dávku rozdělí dle definovaného počtu vláken. Každému vláknu třídy `DQNWorker` přidělí tedy potřebné data a vloží je do seznamu typu tzv. *ThreadPool*, který po svému spuštění spustí všechna vlákna najednou a pak počká na dokončení všech vláken. Tyto vlákna pak vždy po zpracování své dávky zkušeností vloží své nově aktualizované DQN data do datové struktury třídy `DQNNeuralNetNeurons` společné pro všechna vlákna. V této struktuře se po dokončení práce všech vláken provede zprůměrování dat dle počtu vláken.

4.3.2 Master-Slave model

Tento model se podobá předchozímu způsobu paralelizace pomocí vláken s tím, že také rozděluje dávku dat zpracování, ale místo vláknům daný díl dat posílá ji přes síťovou komunikaci patřičnému zpracovateli. Model byl jinak tedy vytvořen dle předchozího návrhu s dříve již dříve popsaným rozhraním.

V tomto modelu tedy operuje hlavní server *Master*, kde běží prostředí s AI agentem a který pak rozesílá rozdělené nasbírané zkušenosti a aktuální DQN svým jednotlivým klientům typu *Slave*, kteří tato svá data zpracují a pošlou zpět a *Master* pak tato data zprůměruje a aktualizuje jimi agentovu DQN.

V tomto modelu jde také využít přímo předchozí způsob paralelizace pomocí vláken, a to konkrétně u klienta *Slave*, který v rámci svého zpracování přidělené dávky dat může i tuto dávku rozdělit a předat postupně jednotlivým vláknům na zpracování. Vše je řešeno zase pomocí třídy `DQNWorker` a pomocí *ThreadPool* v rámci třídy `DQNBackpropagation`. Kdy jsou následně tato zprůměrovaná DQN data z jednotlivých vláken přímo odeslána zpět na server *Master*.

Výsledně je pak dostupná verze pro spuštění lokálně a verze pro spuštění na distribuovaných výpočetních uzlech. Z implementačního hlediska byla v případě tohoto modelu nejvíce kritická implementace spouštěcího rozhraní, které je pak souhrnně popsáno v následující vlastní sekci.

4.3.3 Gorilla DQN model

Tento model představuje již jiný komplikovanější přístup, jak už je zřejmé i z předchozího návrhu. Celkově obsahuje více typů zapojených entit a tím také vzniká větší složitost síťové komunikace, než jak je tomu u předchozího modelu *Master-Slave*. Proto ze začátku vznikaly problémy hlavně s celkovou inicializací a spuštěním jednotlivých částí modelu.

V tomto modelu jde jinak také využít způsob paralelizace pomocí vláken, a to konkrétně u entity typu *Learner*, který má za úkol zpracování přidělené dávky dat, a může i tuto dávku rozdělit a předat postupně jednotlivým vláknům na zpracování a následně zpracovat stejně jako u předchozího modelu *Slave*. S tím rozdílem, že to pak posílá na *ParameterServer*.

4.3.4 Konfigurace paralelních modelů

Konfigurace je jedna ze stěžejních částí celkového paralelního spouštění. Pro tyto účely bylo využito formátu pro serializaci strukturovaných dat YAML, který právě poskytuje čitelnost nejen strojem, ale i člověkem. Dále pak je pomocí knihovny *SnakeYAML* umožněna pohodlná serializace a deserializace mezi Java objekty a YAML dokumenty. Pro konfigurační potřeby byly vytvořeny následující typy konfiguračních tříd.

4.3.5 DQN config

Pro účely DQN konfigurace slouží třída `DQN_Config`, která poskytuje celkovou souhrnnou konfiguraci z hlediska DQN a funguje jako jeden z vstupních inicializačních parametrů pro oba již exportované paralelní modely *Master-Slave* a *Gorilla DQN*. V následujícím výpisu se nachází struktura této konfigurace ve formátu YAML.

```yaml
# DQN config
ann_config:
  activationFunction: !!cs.vsb.bau0025.ANN.actionvationFunctions.Sigmoid {}
  biasInit: 0.0 # počáteční hodnota bias
  lambdaInit: 0.0 # počáteční hodnota lambda
  numberOfHiddenLayers: 2 # počet skrytých vrstev
  numberOfNeuronsInHiddenLayers: [50, 150] # počet neuronů ve skrytých vrstvách
batchSize: 32  # velikost dávky zkušeností pro typ učení STEPS
epsilon: 1.0   # počáteční hodnota epsilon
epsilonStep: null # míra postupného snižování epsilon - v případě null se použije
    výchozí hodnota pro daný typ učení
finalEpsilon: 0.02 # konečná hodnota epsilon
gamma: 0.99 # hodnota proměnné gamma
learningRate: 0.001 # hodnota proměnné learning rate
learningType: EPISODES # vybraný typ učení (STEPS, EPISODES, CYCLES)
maxEpoch: 50000.0 # maximální počet epoch
numberOfThreads: 8 # počet využitých vláken
annCopyConst: 10000 # počet epoch pro zkopírování Target network pro typ učení
    STEPS
replayMemoryMaxSize: 40000 # maximální velikost Replay memory pro typ učení STEPS
replayMemoryMinSize: 1000 # minimální velikost Replay memory pro typ učení STEPS
```

Listing 4.2: YAML DQN config

4.3.6 Aeron config

Další nutností konfigurace je konfigurace knihovny Aeron [19]. Z hlediska paralelních modelů je v ní nutné definovat hlavně porty jednotlivých entit modelu a pak jednotlivé *hostname* všech použitých výpočetních uzlů použitých pro paralelní výpočet. Struktura této konfigurace se mírně u jednotlivých modelů liší dle definovaných entit, a proto existují dvě konfigurační třídy. Třída `AeronConfig` pro model *Master-Slave* a třída `AeronConfigGorilla` pro model *Gorilla DQN*. V následujícím výpisu se nachází struktura těchto dvou typů konfigurací ve formátu YAML.

```yaml
# Aeron Master-Slave config
master: 40450 # port uzlu Master
slaves: [ 40451, 40452, 40453 ] # porty uzlů Slave
nodes: [ cn10, cn2, cn25, cn6 ] # doplnit hostname alokovaných výpočetních uzlů
path: './master_slave_output' # cesta k uložení výstupních souborů
verboseOutput: false # podrobné logování
```

Listing 4.3: YAML Master-Slave Aeron config

```yaml
# Aeron Gorilla DQN config
actors: [ 40450, 40451 ] # porty uzlů Actor
learners: [ 40460, 40461 ] # porty uzlů Learner
globalMemory: 40440 # port uzlu GlobalMemory
parameterServer: 40441 # port uzlu ParameterServer
nodes: [ r2i0n0, r4i3n14, r4i4n0, r4i4n5 ] # doplnit hostname alokovaných výpočetn
    ích uzlů
path: './gorillaDQN_output' # cesta k uložení výstupních souborů
GLOBAL_MEMORY_BATCH: 512 # velikost dávky vybírané z GlobalMemory
verboseOutput: false # podrobné logování
```

Listing 4.4: YAML Gorilla DQN Aeron config

4.4 HPC spouštění

Pro potřeby spouštění na dístribuovaných výpočetních uzlech bylo nejdůležitějším prvkem samotné spouštěcí rozhraní exportované aplikace. Tato aplikace totiž musí mít dostatečně univerzální spouštěcí rozhraní, aby ji bylo možné inicializovat a spustit na daném výpočetním uzlu, tak aby měla přidělený svůj vlastní unikátní uzel a zároveň, aby měla povědomí o ostatních uzlech dle svojí potřeby. Jedním z problémů tohoto spouštění je, že po alokaci určitého počtu výpočetních uzlů s určitým počtem jader v případě distribuovaných výpočetních uzlů na IT4Innovations [21], dále IT4I, se pak při spouštění jednotlivé alokované uzly nespouštějí podle daného pořadí, ale náhodně

dle aktuální možnosti spuštění daného uzlu. Kvůli této komplikaci muselo být vytvořeno spouštěcí rozhraní na tyto podmínky připravené.

4.4.1 Popis spouštěcího rozhraní

Toto rozhraní je u obou paralelních modelů velice podobné. Obsahuje vždy možnost spuštění lokálně, kdy stačí do konfiguračního YAML dokumentu, napsat pouze čísla portů dle počtu potřebných výpočetních entit a následný tzv. *hostname*, tedy unikátní název daného uzlu, se doplní jako *localhost*, takže není potřeba vyplňovat *hostname* pole *nodes*. V případě spuštění na HPC, musí být v *nodes* poli pro *hostname* uzlů přímo zadány *hostname* všech zúčastněných alokovaných uzlů, které také ovšem musí korespondovat s celkovým počtem uvedených portů.

Celý princip pak spočívá v tom, že se jako dva argumenty pro exportovaný spustitelný soubor JAR vloží dva již uvedené YAML konfigurační soubory - DQN_Config a AeronConfig (případně AeronConfigGorilla) a potom v případě HPC se vloží třetí *hostname* argument, který je získán proměnou `$HOSTNAME`. Poté se v pořadí spuštění tohoto jednoho daného JAR souboru, který postupně spouští každý uzel zvlášť, se postupně vytváří tzv. *zamykací soubory* pro jednotlivé zapojené výpočetní entity, což je tedy prostý textový soubor s názvem nebo pořadí uzlu. Každý uzel se tak při svém spouštění, nejdříve podívá, zda už nějaký uzel tuto výpočetní entity již nezamkl, tedy zda existuje zamčený soubor a poté ji v případě, že neexistuje, tak ji sám inicializuje nebo se přesune na inicializaci další možné výpočetní entity. Toto pořadí vytváření a zamykání entit je dáno dle důležitosti, takže například se jako první spustí a zamkne *Master* uzel a až pak jednotlivé *Slave* uzly.

4.4.1.1 Rozdíly spouštění paralelních modelů

U modelu *Master-Slave* se tedy z prvního uzlu, který spustí JAR soubor stává *Master*. Tento uzel pak si na svém *hostname* dle konfigurace spustí na master portu svou instanci a ostatní *hostname* v poli *nodes* v YAML konfiguračním souboru pro *Aeron* si přiřadí jako pole možných uzlů *Slave*. Následně před spuštěním zapíše ještě do zamykacího textového souboru svou celou adresu ve tvaru `HOSTNAME:PORT`. Následně se pak spustí s danou adresou a *DQN_Config* souborem. Další spuštěný uzel se pak nejdříve podívá, zda tento master uzamknutý soubor již existuje a pokud ano, tak si z toho souboru načte zapsanou adresu uzlu *Master*, kterou použije pro připojení tomuto uzlu. Dále si v YAML konfiguračním souboru pro *Aeron* v poli *nodes* najde svůj *hostname*, přidělí si port z pole *slaves* podle pořadí spuštění a vytvoří zamykací soubor s indexem vybraného portu z pole *slaves*. Pak se spustí se svou adresou a *DQN_config* souborem. Poté po inicializaci všech entit probíhá výpočet a síťová komunikace mezi těmito uzly.

U modelu *Gorilla DQN* se pak z prvního uzlu, který spustí JAR soubor stává *ParameterServer*. Tento uzel pak musí v rámci své zamykaní vytvořit nový "zamknutý"YAML konfigurační soubor pro Aeron, který obsahuje již celé další rozdělené schéma následného spouštění. Musí již všem zbývajícím

hostname uzlů v poli *nodes* přiřadit správnou entitu a port zase ve tvaru `HOSTNAME:PORT`. Po tomto přiřazení zapíše tento definovaný YAML soubor a spustí se s jeho adresou a *DQN_Config* souborem. Další spuštené uzly si po zjištění, že již tento zamknutý konfigurační YAML existuje, tento soubor načtou a najdou si v něm podle svého *hostname* svou již přidělenou adresu a spustí se zase s touto svojí adresou a *DQN_Config* souborem. Poté zase po inicializaci všech entit probíhá výpočet a síťová komunikace mezi těmito uzly.

4.4.2 Postup spouštění na IT4I HPC

Pro spuštění je potřeba si nejdříve také třeba připravit tzv. *Jobscript* pro spouštění dávkových úloh v tomto frontovém systému PBS a dále pak přesunout všechny soubory pro spuštění do podadresáře projektu v adresáři `SCRATCH`.

Spouštění na distribuovaných výpočetních uzlech IT4I poté probíhá následujícím způsobem.

1. Alokace daného počtu uzlů v interaktivním režimu, nejčastěji pomocí produkční nebo expresní fronty
   - produkční fronta
     `qsub -A [ID PROJEKTU] -q qprod -l select=4:ncpus=36 -I`
   - expresní fronta
     `qsub -q qexp -l select=4:ncpus=24 -I`
2. Zjištění alokovaných uzlů příkazem `sort -u $PBS_NODEFILE`
3. Zapsaní získaných *hostname* všech uzlů (bez domény) do pole `nodes` v YAML konfiguračním souboru pro Aeron
4. Přesunutí všech potřebných spustitelných a konfiguračních souborů do podadresáře projektu v adresáři `SCRATCH`
5. Spuštění běhu na jednotlivých uzlech najednou s definicí daného *Jobscript* souboru
   - Hlavní definice v *Jobscript* souboru s DQN config souborem a Aeron config souborem pro daný paralelní model
     Např. `java -jar ./breakout-master-slave.jar ./aeron-master-slave-config.yaml ./dqn-config.yaml $HOSTNAME » ./output.txt`
   - Následné spuštění na alokovaných uzlech
     Např. `pdsh -w r2i6n14,r2i6n7,r3i6n3,r3i7n2 sh job_aeron.sh`

4.5 Export aplikace

Export aplikace do spustitelného souboru JAR je, jak už bylo zmíněno, realizován s použitím nástroje Apache Maven pomocí pluginu Apache Maven Shade Plugin. Tento plugin vytvoří spustitelný JAR soubor se všemi potřebnými závislostmi a využívá se pak v rámci daných profilů pomocí, kterých mohou být dle potřeby sestaveny jednotlivé části jako samostatně spustitelné aplikace.

Profily pro možné typy sestavení jsou následující:

- **env** - profil k sestavení spustitelné verze s grafickým rozhraním
- **console** - profil k sestavení spustitelné konzolové verze
- **hpc** - profil k sestavení spustitelných verzí obou paralelních modelů

Samotné spuštění sestavení se provadí příkazem: `mvn clean package -P [NÁZEV PROFILU]`. Po úspěšném sestavení se spustitelné JAR soubory nachází ve složce `target`.

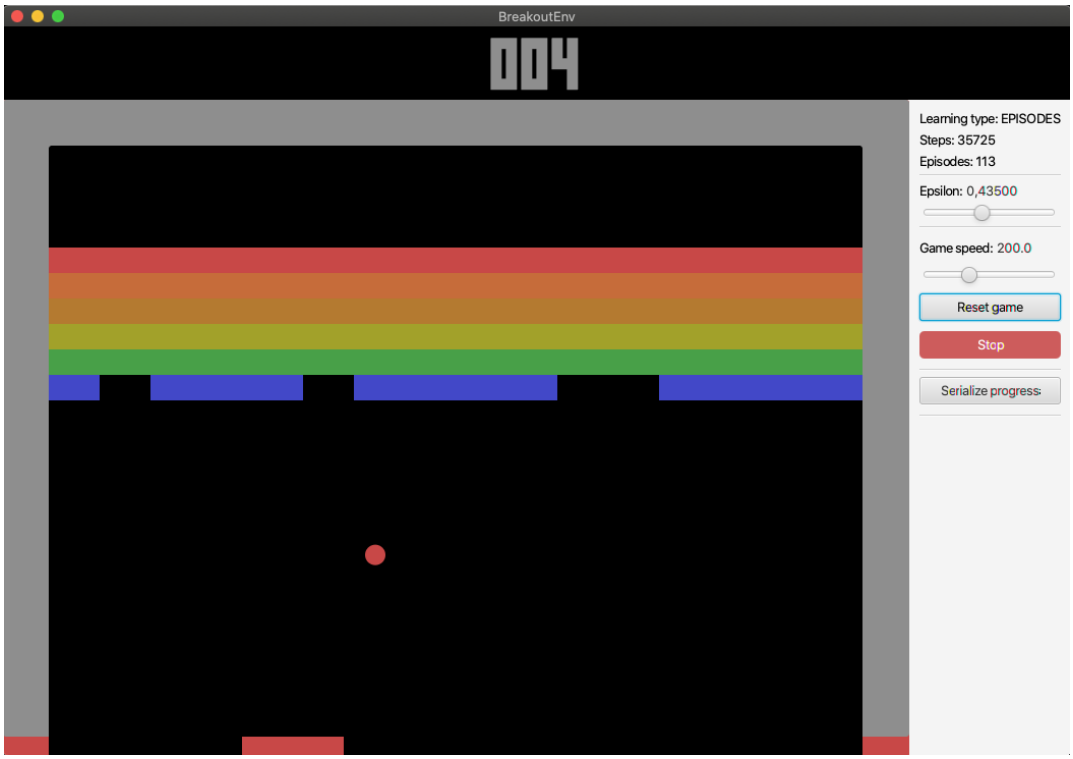
Kapitola 5

Výsledky a testování

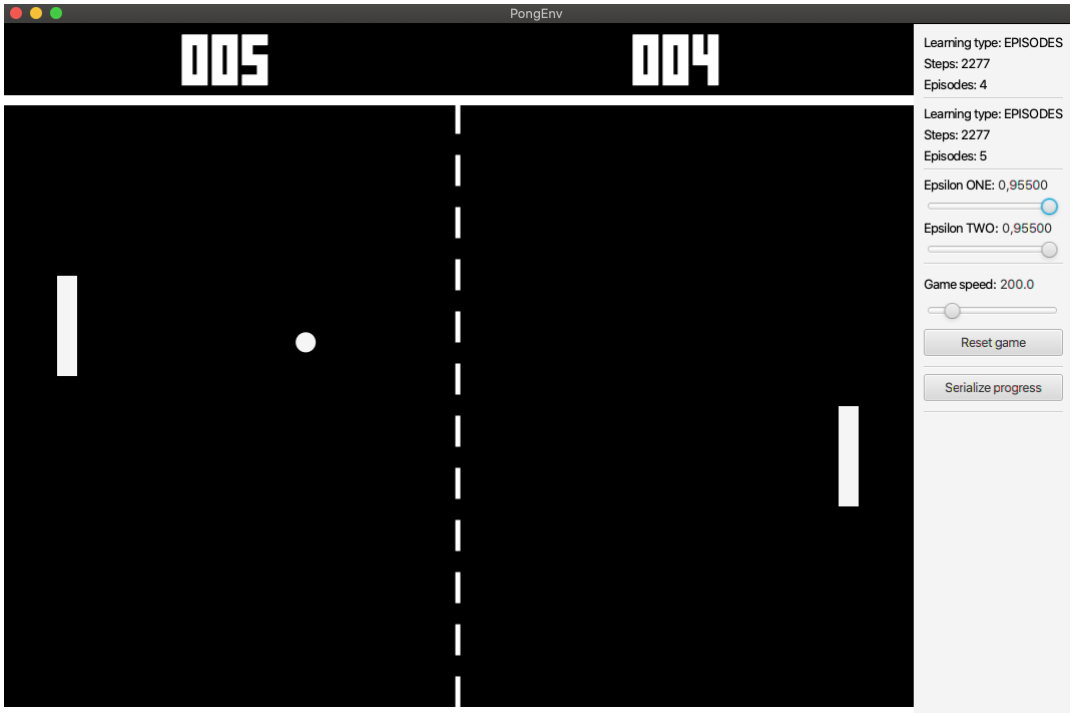
Tato kapitola pojednává o výstupech projektu této práce, a poté se zaměřuje na testování a porovnání jednotlivých typů realizovaných výpočtů s danou konfigurací. Testování bylo prováděno na distribuovaných výpočetních uzlech IT4I s využitím superpočítače Salomon.

5.1 Výsledné grafické rozhraní aplikace

Hlavním výstupem je tedy spustitelná JavaFX [17] aplikace, která obsahuje menu s výběrem testovacího prostředí pro simulaci a experimenty. Po výběru daného prostředí je potřeba zadat patřičnou konfiguraci realizovanou konfiguračním YAML souborem reprezentujícím třídu `DQN_Config`. Poté po načtení konfigurace je spuštěno dané prostředí a zahájeno Deep Q-learning učení [10]. Toto učení je možné zvolit jako postupné, kdy je možné vidět celý průběh interakce agenta s prostředím v rámci učení v panelu s grafickým náhledem prostředí a nebo lze zvolit varianta pro rychlé učení na pozadí, které po dosažení ukončovacích kritérii vrátí již naučeného agenta s nímž pak spustí tento grafický náhled. Samotné spuštěné rozhraní prostředí tedy zahrnuje, jak hlavní panel s běžícím grafickým náhledem prostředí a pak také panel s aktuálními informacemi o průběhu a s menší dodatečnou konfigurací tohoto průběhu učení.



Obrázek 5.1: Grafické rozhraní prostředí Breakout [13]



Obrázek 5.2: Grafické rozhraní prostředí Pong [14]

5.2 Testování s využitím HPC

Testování bylo tedy prováděno za stejné konfigurace na distribuovaných výpočetních uzlech superpočitače Salomon spadajících pod IT4I. Ovšem toto testování bylo nakonec prováděno jenom nad prostředím Breakout [13], kde operuje jeden agent, jelikož druhé prostředí Pong [14], kde operují dva agenti, se pro účely paralelizace podle předběžných návrhů ukázalo jako velice náročné na implementaci a pro účely této práce nebylo celkově nakonec časově zcela realizovatelné. Navíc také prostředí Breakout zcela vyhovuje podmínkám pro porovnávání.

V rámci tohoto testování s využitím paralelizace bylo následně zjištěno, které zvolené typy učení jsou vhodné pro které paralelní modely. Ukázalo se tedy, že pro model *Master-Slave* se značné nehodí typ učení po jednotlivých časových krocích - *STEPS*, který nedosahoval víceméně žádných použitelných výsledků. Ostatní typy EPISODES a CYCLES se ukázaly jako vhodné s patřičnými výsledky, ovšem lepších výsledků pak dosahoval typ učení EPISODES. Na druhou stranu se ukázalo, že pro paralelní model *Gorilla DQN* nejlépe vyhovuje typ učení STEPS. Typy učení EPISODES a CYCLES se zase ukázaly jako nevhodné pro tento model.

Dále u způsobu paralelizace pomocí vláken bylo po sléze zjištěno, že je vyhovující jenom pro typy učení EPISODES a CYCLES. Tyto typy učení pracují nárazově s mnohem většími dávkami zkušeností než typ STEPS, pro který je toto samotné spouštění vláken více časově náročnější nebo velice časově náročné, než klasický sekvenční přístup, jelikož se toto spouštění provádí v každém časovém kroku, což představuje velkou zátěž pro toto spouštění vláken a dochází víceméně k vyčerpání prostředků.

5.2.1 Způsob porovnávání

Pro účely porovnání z hlediska měřených parametrů bylo využito následujících měřených údajů:

- čas dokončení výpočtu
- počtu proběhlých epizod (značí počet restartovaných kol prostředí, tedy čím méně, tím lépe)
- suma odměny za jednu epizodu

Dále byly definovány následující ukončovací parametry:

- počet epoch
- finální hodnota proměnné Epsilon

Jak už bylo také naznačeno, nelze právě jednoduše porovnávat všechny zvolené typy učení mezi sebou, jelikož pracují trochu s jinými modely a přístupy k paralelizaci. Byly tak hlavně porovnány jednotlivé typy učení se svými přístupy k paralelizaci, kdy hlavním záměrem bylo zjištění toho jak se změní měřené hodnoty výpočtu po přidání možných zdrojů. Tedy například přidání více vláken

nebo více výpočetních uzlů a poté bylo určeno která konfigurace je nejlepší. Na závěr byly, ale také některé typy učení v určitém případě podobné navzájem porovnány.

5.2.2 Hlavní použitá konfigurace DQN

Pro účely porovnávání výpočtů byla stanovena tato potřebná konfigurace klíčových DQN parametrů společných pro všechny typy učení.

Tabulka 5.1: Použitá konfigurace DQN

Maximální počet epoch	50000
Počáteční bias	0.0
Počáteční lambda	0.0
Learning rate	0.001
Počet skrytých vrstev	2
Počet neuronů ve skrytých vrstvách	[50, 150]
Počet vstupů	4
Počet výstupů	3
Počáteční Epsilon	1.0
Konečné Epsilon	0.2
Discount faktor Gamma	0.99

5.2.3 Porovnání paralelizace typu učení STEPS

Tento typ učení bylo možné porovnat z hlediska sekvenčního, kdy se měnila velikost dávky zkušeností, která se náhodně vytahuje z Replay memory při každém časovém kroku a dále pak s využitím modelu Gorilla DQN, který je určen hlavně pro tento typ učení.

Pro tento typ učení musela být poskytnuta rozšířená hlavní konfigurace o dané parametry, kdy se jedná o tyto parametry.

Tabulka 5.2: Dodatečná použitá konfigurace DQN

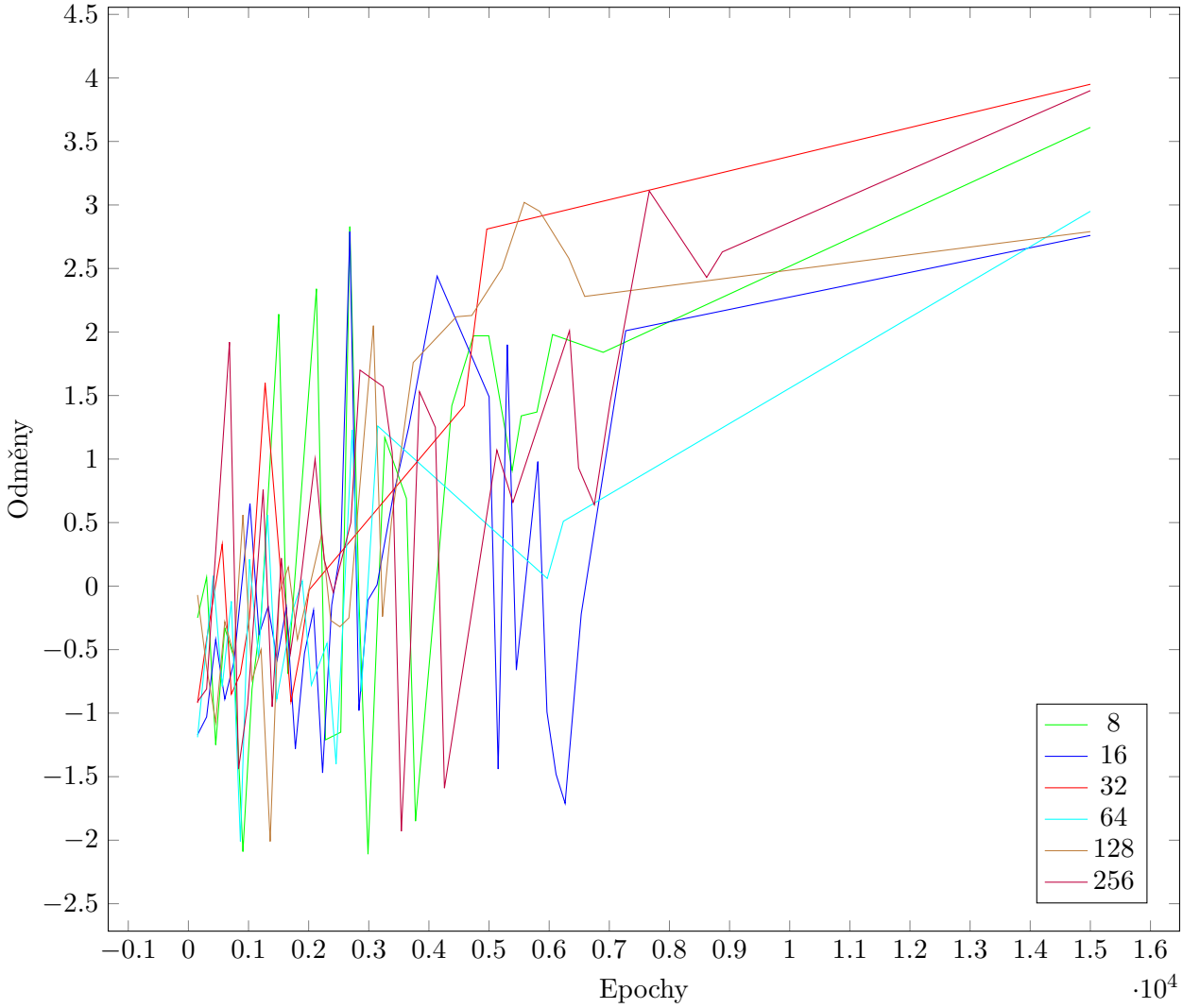
Minimální velikost Replay memory	1500
Maximální velikost Replay memory	40000
Konstanta kopírování Target DQN	10000

5.2.3.1 Porovnání s různou velikostí dávky zkušeností

Zde jsou uvedeny naměřené hodnoty pro danou velikost dávky zkušeností a jejího vlivu na celkový čas dokončení výpočtu, počtu epizod a odměnu dosahovanou v jednotlivých epizodách.

Tabulka 5.3: Vliv velikosti dávky zkušeností na dobu běhu výpočtu a počet epizod

Velikost dávky	Doba běhu (s)	Počet epizod
8	30,62	27
16	58,69	33
32	94,60	13
64	226,85	19
128	455,80	28
256	915,30	31



Obrázek 5.3: Graf dosažených odměn jednotlivých epizod v rámci epoch typu učení STEPS

Z těchto naměřených údajů můžeme usoudit, že postupně se zvětšováním velikosti dávky zkuše-

ností stoupá doba běhu výpočtu, ale zároveň z stoupá kvalita učení, tedy počet epizod klesá. Ovšem potom postupně od dávky 32 vidíme i postupný nárůst počtu epizod a tedy postupnou degradaci učení. Dále pak z grafu dosažených odměn jednotlivých epizod vidíme, že dosažená odměna stoupá tedy u všech velikostí dávek zkušeností nakonec velice podobným lineárním způsobem, což značí, že agent se s využitím své DQN naučí efektivně operovat v daném prostředí s využitím všech velikostí těchto dávek zkušeností.

Pak z hlediska nejlepší konfigurace se nejlépe jeví dávka zkušeností o velikosti 32, která dosahuje nejmenšího počtu epizod v rámci učení a také nejvyšší konečné odměny za epizodu.

5.2.3.2 Porovnání v rámci modelu Gorilla DQN

Zde jsou uvedeny naměřené hodnoty pro danou velikostí dávky zkušeností a v rámci daného počtu použitých uzlů *Actor* a *Learner*. Můžeme pozorovat vliv těchto nastavení na celkový čas dokončení výpočtu, na počet epizod a na odměnu dosahovanou v jednotlivých epizodách.

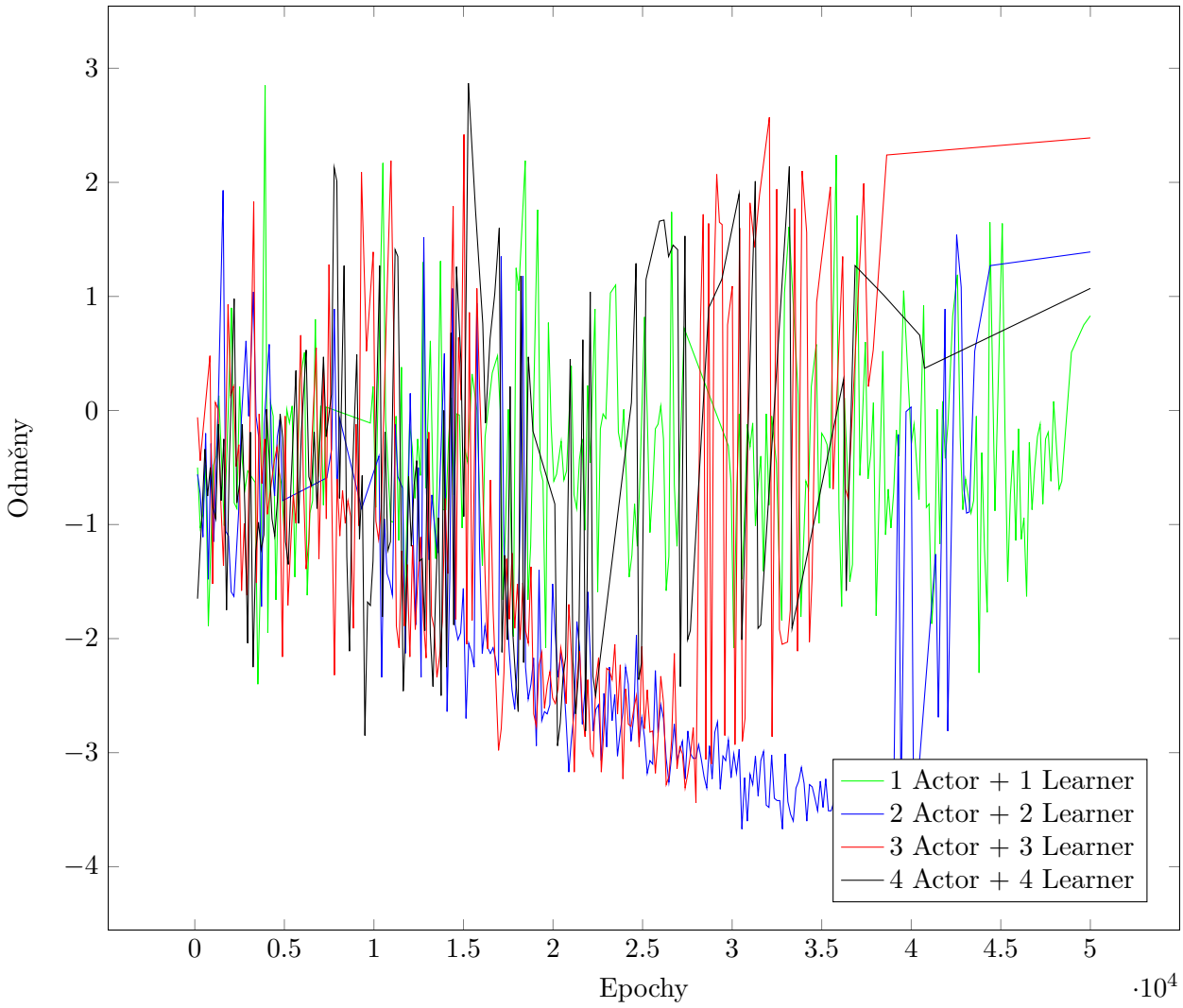
Pro porovnání odměny dosahované v jednotlivých epizodách s pomocí tohoto paralelního modelu byla pak vybrána dávka zkušeností o velikosti 512, která se v případě naměřených hodnot v následujících tabulkách ukázala jako nejefektivnější.

Tabulka 5.4: Doba běhu výpočtu (s) s využitím modelu Gorilla DQN s daným počtem uzlů Actor (A) a Learner (L) a danou velikostí dávky zkušeností

Velikost dávky zkušeností	1 A + 1 L	2 A + 2 L	3 A + 3 L	4 A + 4 L
32	122,05	122,01	122,15	130,49
64	122,02	122,02	126,59	128,90
128	122,05	121,89	122,01	124,02
256	122,06	121,07	120,89	122,02
512	121,73	117,61	116,99	119,01
1024	120,84	120,83	118,84	119,33

Tabulka 5.5: Počet epizod s využitím modelu Gorilla DQN s daným počtem uzlů Actor (A) a Learner (L) a danou velikostí dávky zkušeností

Velikost dávky zkušeností	1 A + 1 L	2 A+ 2 L	3 A + 3 L	4 A + 4 L
32	242	266	268	278
64	244	242	266	294
128	256	245	267	281
256	282	278	298	304
512	251	198	197	240
1024	258	239	267	298



Obrázek 5.4: Graf dosažených odměn jednotlivých epizod v rámci epoch typu učení STEPS s využitím paralelního modelu Gorilla DQN

Z těchto naměřených údajů můžeme usoudit, že na rozdíl od předchozího přístupu bez použití modelu *Gorilla DQN*, postupně se zvětšováním velikosti dávky zkušeností doba běhu výpočtu se spíše zkracuje, ale kvalita učení daná počtem epizod se tolik neliší. Pak jsou zde na porovnání také naměřené hodnoty dle počtu použitých uzlů typu *Actor* a *Learner*. V tomto případě můžeme vidět, že dochází k tomu, že s postupným větším počtem těchto použitých typů uzlů dochází k zlepšení kvality učení, ale zároveň potom již s moc velkým počtem těchto použitých typů uzlů (4 Actor + 4 Learner) dochází zase k úpadku kvality učení pravděpodobně kvůli náročnosti vzájemné komunikace uzlů a celkové synchronizace modelu.

Z hlediska pak nejefektivnější konfigurace vychází pak nejlépe již zmíněná dávka zkušeností o velikosti 512 s počtem 3 Actor uzlů a 3 Learner uzlů daná dle počtu epizod a velikosti konečné

dosažené odměny jednotlivých epizod v průběhu učení.

5.2.4 Porovnání paralelizace typu učení EPISODES

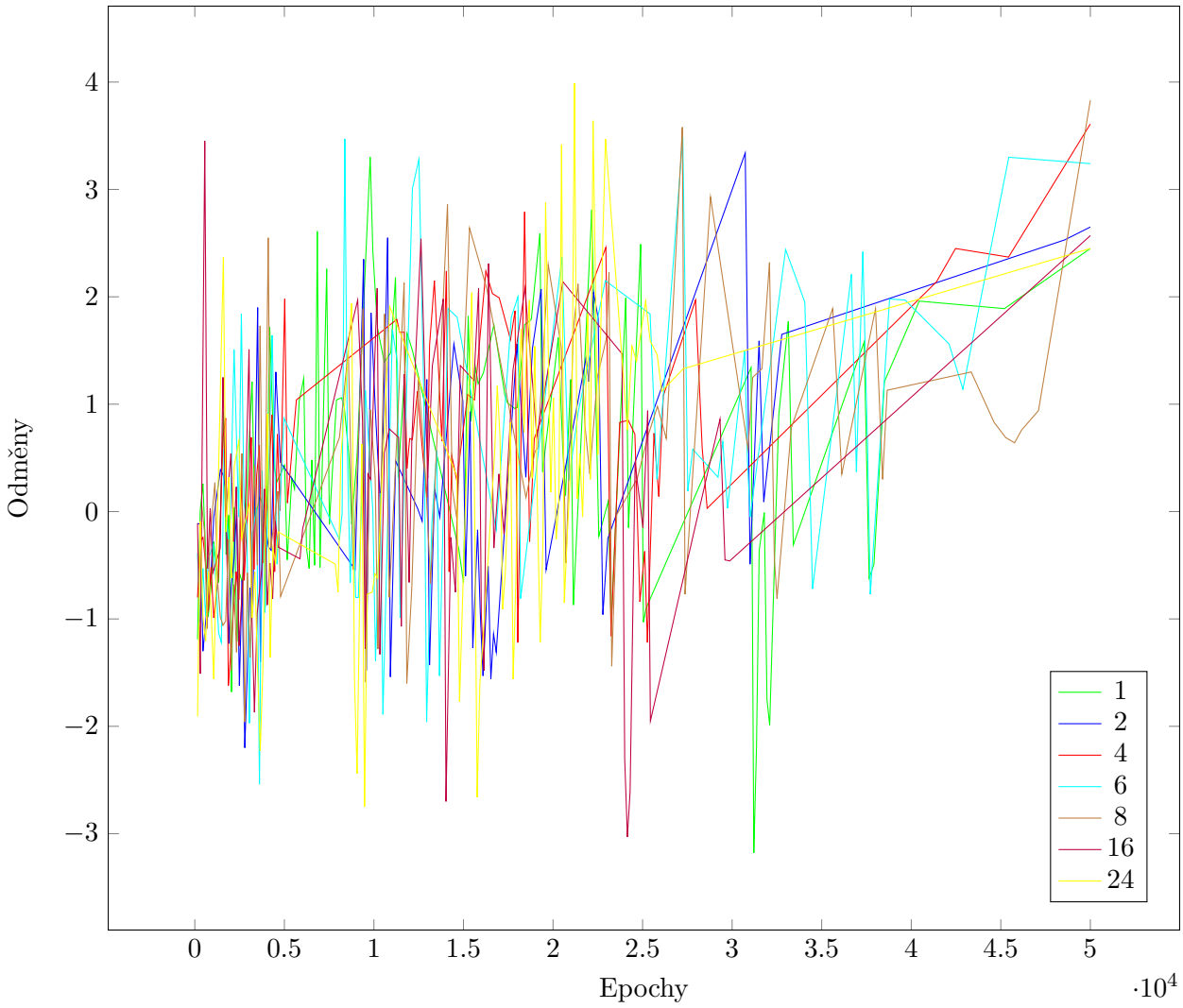
Tento typ učení bylo možné porovnat s použitím různého počtu vláken a tedy CPU jader a dále pak s využitím modelu Master-Slave, který je hlavně určen pro tento typ učení.

5.2.4.1 Porovnání s využitím daného počtu vláken

Zde jsou uvedeny naměřené hodnoty pro jednotlivé počty použitých vláken a poté hodnoty jejich vlivu na celkový čas dokončení výpočtu, na počet epizod a na odměnu dosahovanou v jednotlivých epizodách.

Tabulka 5.6: Vliv počtu vláken na dobu běhu a počet epizod

Počet vláken	Doba běhu (s)	Počet epizod
1	22,50	91
2	16,08	88
4	12,34	90
6	12,03	107
8	12,51	111
16	14,28	119
24	21,68	124



Obrázek 5.5: Graf dosažených odměn jednotlivých epizod v rámci epoch s použitím typu učení EPISODES a daným počtem vláken

Z naměřených údajů můžeme pozorovat, že s větším počtem vláken se snižuje doba běhu výpočtu, ale kvalita učení daná počtem epizod zůstává víceméně na stejné úrovni. Ovšem poté s již velkým počtem vláken dochází postupně k delší době běhu výpočtu a úpadku kvality učení. Tento celkový úpadek je dán kvůli tomu, že s větším počtem vláken se zmenšuje jim přidělovaná dávka a proto se pak od určitého počtu stává časově náročnější samotné spouštění těchto vláken, než sekvenční zpracování této dávky zkušeností. Také pak samotná kvalita DQN dat se kvůli synchronizaci těchto dat z většího počtu vláken pomocí průměrování se stává horší. Dále pak z grafu dosažených odměn jednotlivých epizod vidíme, že dosažená odměna nakonec stoupá u všech výpočtu s daným počtem vláken velice podobným tempem, což značí, že agent se s využitím své DQN naučí efektivně operovat v daném prostředí s různým počtem vláken.

Z hlediska nejefektivnější konfigurace pak vychází počet vláken od 4 do 8 dle doby běhu výpočtu, kvality učení a dosažené konečné odměny za jednotlivé epizody.

5.2.4.2 Porovnání v rámci modelu Master-Slave

Zde jsou uvedeny naměřené hodnoty s využitím daného počtu použitých uzlů *Slave* a také dle počtu použitých vláken v rámci těchto uzlů. Můžeme tedy pozorovat vliv této konfigurace na celkovou dobu běhu výpočtu, na počet epizod a na odměnu dosahovanou v jednotlivých epizodách.

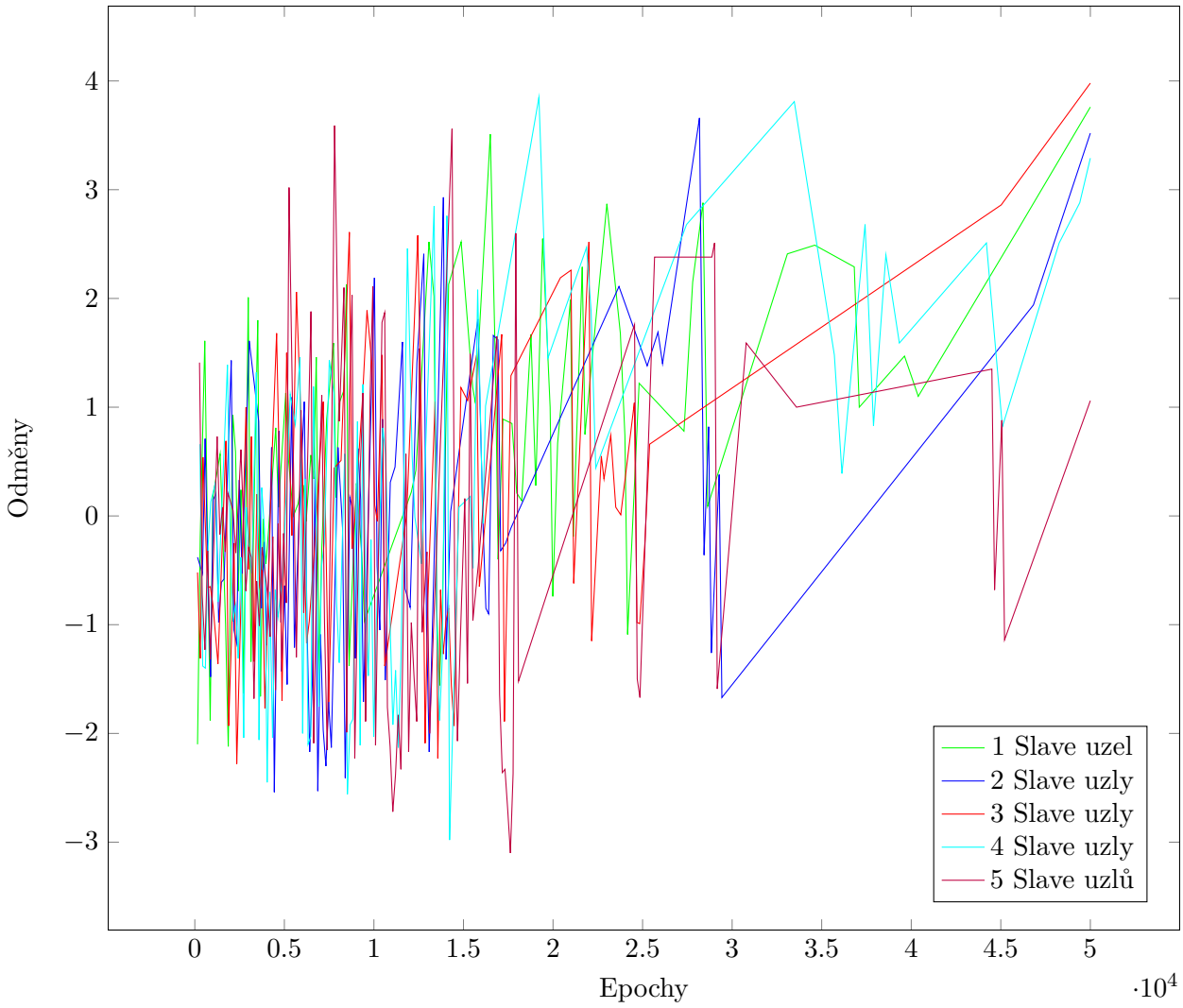
Pro porovnání odměny dosahované v jednotlivých epizodách s pomocí toho paralelního modelu bylo pak vybráno zpracování jedním vláknem na jeden uzel, což se v případě naměřených hodnot v následujících tabulkách ukázalo jako nejefektivnější přístup.

Tabulka 5.7: Doba běhu výpočtu (s) s využitím modelu Master-Slave s daným počtem výpočetních uzlů a vláken

Počet vláken	1 Slave uzel	2 Slave uzly	3 Slave uzly	4 Slave uzly	5 Slave uzlů
1	66,66	59,61	52,34	60,12	70,58
2	62,70	60,80	54,82	64,20	82,96
4	67,85	61,80	71,67	66,70	98,15
6	67,46	65,47	96,57	69,39	113,39
8	72,44	68,29	113,87	87,74	113,77
16	87,08	84,77	84,02	113,54	122,16
24	112,32	103,91	89,47	128,62	128,84

Tabulka 5.8: Počet epizod s využitím modelu Master-Slave s daným počtem výpočetních uzlů a vláken

Počet vláken	1 Slave uzel	2 Slave uzly	3 Slave uzly	4 Slave uzly	5 Slave uzlů
1	94	89	85	101	106
2	93	97	92	105	141
4	108	92	123	107	173
6	108	109	166	116	180
8	118	115	194	137	184
16	138	143	133	194	200
24	171	165	143	200	200



Obrázek 5.6: Graf dosažených odměn jednotlivých epizod v rámci epoch s použitím typu učení EPISODES a paralelního modelu Master-Slave

Z těchto naměřených údajů můžeme usoudit, že tento paralelní model *Master-Slave* se na rozdíl od předchozího přístupu s použitím vláken liší hlavně v době běhu výpočtu. Ostatní parametry, tedy počet epizod a dosažená odměna v jednotlivých epizodách jsou velice podobné nebo i lehce lepší. Můžeme však pozorovat, že samotná doba výpočtu se s přidáváním vláken zvětšuje a zároveň díky možnosti přidáváním výpočetních *Slave* uzlů se tato doba zase zkracuje, ale také do určitého počtu těchto uzlů, kdy se potom jak doba výpočtu, tak i kvalita učení začíná zhoršovat. K zhoršování výsledků dochází kvůli menší dávce zkušeností, která je přidělována jednotlivým *Slave* uzlům, jelikož se celková paměť zkušeností, tedy *Replay memory*, jenž obsahuje získané zkušeností za jednu epizodu prostředí dělí větším dílem dle počtů těchto uzlů. Poté může být právě ještě v rámci *Slave* uzlu také rozdělena na menší dávky pro jednotlivá vlákna, a tak se z hlediska čím dál menší přidělené dávky

stává více časové náročné tyto vlákna spouštět. Dále pak je větší problém se synchronizací DQN dat, která se při sloučení ze všech uzlů průměrují a tak určitým způsobem ztrácí svou kvalitu a pak se toto stává s přibývajícím počtem těchto *Slave* uzlů také časově náročnější. Zase však z hlediska dosahované konečné odměny za epizodu vidíme, že všichni agenti se s využitím své DQN také úspěšně naučí operovat v daném prostředí.

Pak v případě co nejefektivnější konfigurace vychází nejlépe použití 3 *Slave* uzlů s jedním vláknem, jak je vidět z předchozích tabulek naměřených hodnot a grafu dosahované odměny za epizodu.

5.2.5 Porovnání paralelizace typu učení CYCLES

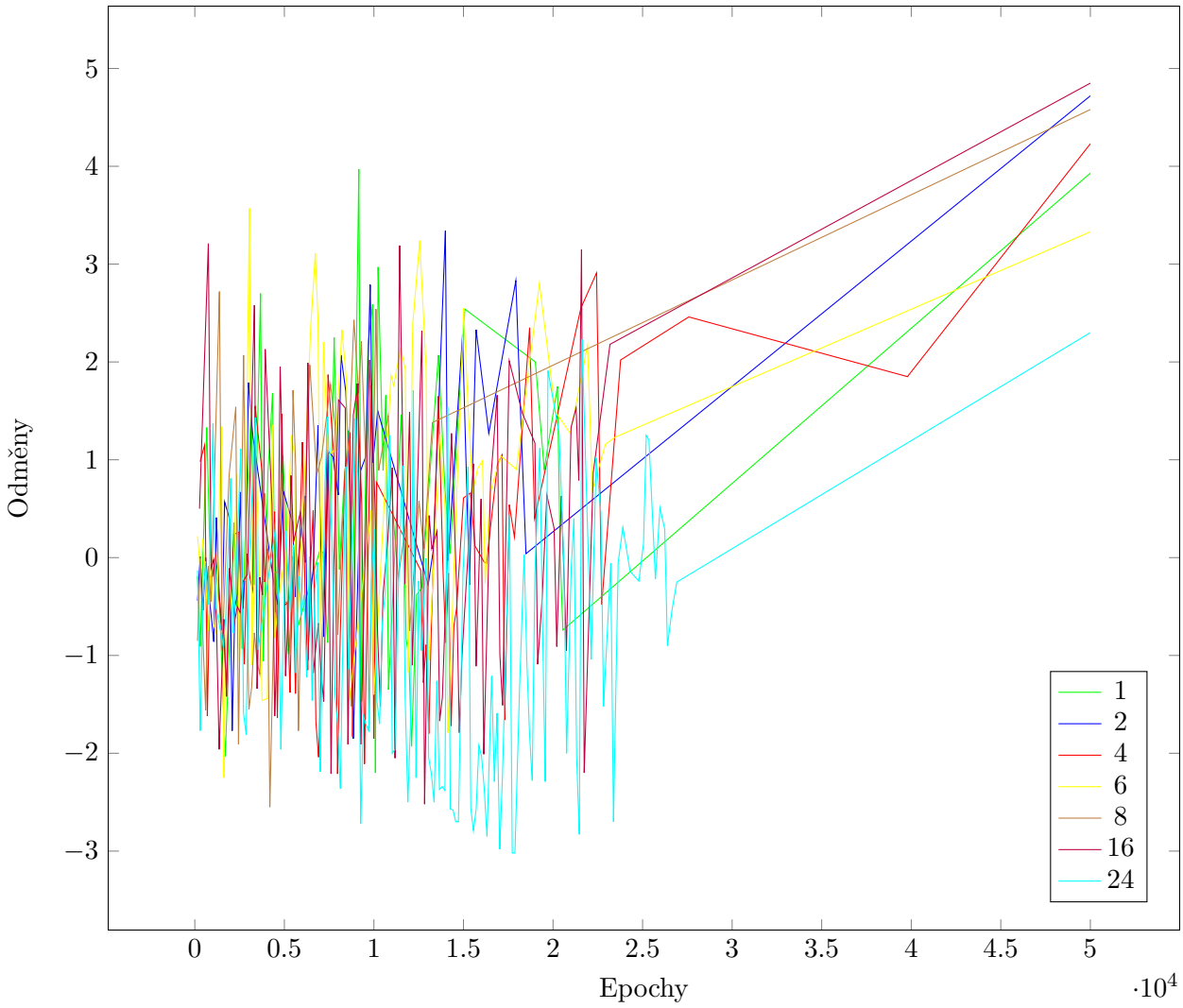
Tento typ učení bylo stejně jako v případě typu učení *EPISODES* možné porovnat s použitím různého počtu vláken nebo tedy CPU jader a dále pak s využitím modelu Master-Slave, který je hlavně určen pro tento typ učení.

5.2.5.1 Porovnání s využitím daného počtu vláken

Zde jsou uvedeny naměřené hodnoty pro jednotlivé počty použitých vláken a poté hodnoty jejich vlivu na celkový čas dokončení výpočtu, na počet epizod a na odměnu dosahovanou v jednotlivých epizodách.

Tabulka 5.9: Porovnaní času dokončení výpočtu a počtu epizod s daným počtem vláken

Počet vláken	Doba běhu (s)	Počet epizod
1	22,29	61
2	17,55	57
4	14,93	63
6	17,98	72
8	18,69	54
16	31,41	95
24	42,41	134



Obrázek 5.7: Graf dosažených odměn jednotlivých epizod v rámci epoch s použitím typu učení CYCLES a vláken

Z naměřených údajů můžeme pozorovat, že s větším počtem vláken se zase stejně jako u *EPISODES* snižuje doba běhu výpočtu, ale zase do určitého počtu vláken, kdy potom už tato doba výpočtu začíná zase stoupat. Tato degradace je dána kvůli tomu, že s větším počtem vláken se zmenšuje jim přidělovaná dávka a proto se pak od určitého počtu stává časově náročnější samotné spouštění těchto vláken, než sekvenční zpracování této dávky zkušeností. Kvalita učení daná počtem epizod se pak pohybuje podobně, jako doba běhu, ale víceméně konsoliduje tak na stejné úrovni a spíše má tendenci se zhoršovat kvůli náročnější synchronizaci dat dle počtu použitých vláken. Dále pak z grafu dosažených odměn jednotlivých epizod vidíme, že dosažená odměna nakonec stoupá u všech výpočtů s daným počtem vláken velice úspěšným lineárním tempem, což značí, že agent se s využitím své DQN naučí efektivně operovat v daném prostředí s různým počtem vláken.

Z hlediska nejefektivnější konfigurace pak vychází počet vláken od 2 do 8 dle doby běhu výpočtu, kvality učení a dosažené konečné odměny za jednotlivé epizody, jelikož potom již tyto naměřené výsledky začínají degradovat.

5.2.5.2 Porovnání v rámci modelu Master-Slave

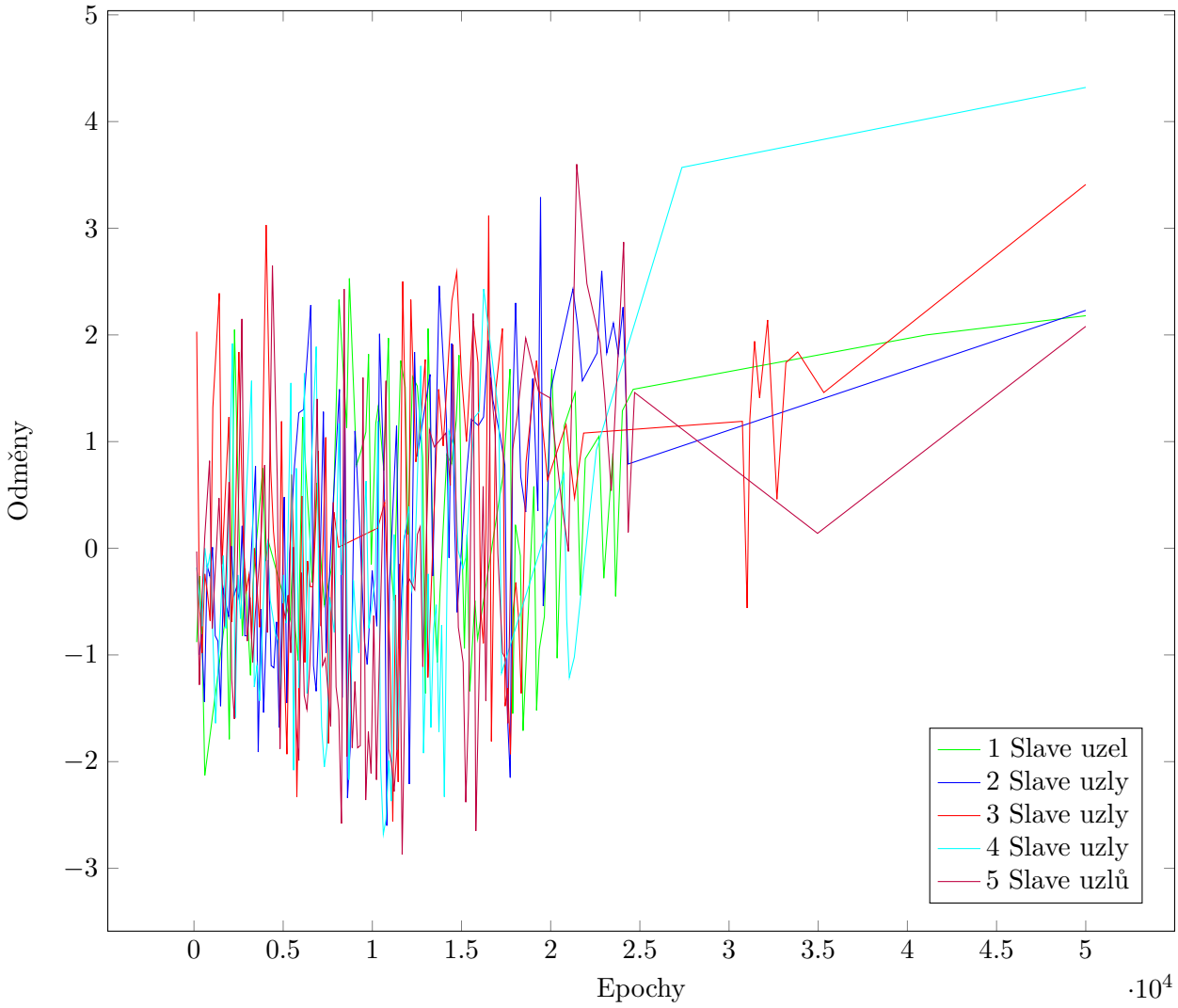
Zde jsou uvedeny naměřené hodnoty s využitím daného počtu použitých uzlů *Slave*, které mohou také využívat pro účely svého výpočtu různý počet vláken, jenž má dopad na celkový úspěch výpočtu. Můžeme tedy pozorovat vliv různého nastavení této konfigurace na celkovou dobu běhu výpočtu a na kvalitu učení udávanou počtem epizod a odměnou dosahovanou v jednotlivých epizodách. Pro porovnání odměny dosahované v jednotlivých epizodách s pomocí tohoto paralelního modelu bylo pak vybráno zpracování jedním vláknem na jeden uzel, což se v případě naměřených hodnot v následujících tabulkách z celkového počtu použitých uzlů *Slave* ukázalo jako nejefektivnější přístup.

Tabulka 5.10: Doba běhu výpočtu (s) s využitím modelu Master-Slave s daným počtem výpočetních uzlů a vláken

Počet vláken	1 slave uzel	2 slave uzly	3 slave uzly	4 slave uzly	5 slave uzlů
1	179,82	169,10	165,39	168,22	177,88
2	175,36	159,12	180,55	175,65	178,71
4	170,39	157,25	182,21	182,12	183,66
8	184,45	187,82	190,09	188,45	187,57
6	186,91	186,08	186,43	185,36	183,34
16	193,89	201,02	194,96	200,37	204,96
24	208,49	212,72	211,52	216,88	220,29

Tabulka 5.11: Počet epizod s využitím modelu Master-Slave s daným počtem výpočetních uzlů a vláken

Počet vláken	1 slave uzel	2 slave uzly	3 slave uzly	4 slave uzly	5 slave uzlů
1	89	102	97	90	107
2	96	98	139	114	122
4	114	103	151	175	256
6	98	212	217	234	280
8	159	133	251	219	217
16	167	250	232	285	300
24	251	227	283	317	274



Obrázek 5.8: Graf dosažených odměn jednotlivých epizod v rámci epoch s použitím typu učení CYCLES a paralelního modelu Master-Slave

Z těchto naměřených údajů můžeme usoudit, že tento paralelní model *Master-Slave* se na rozdíl od předchozího přístupu s použitím vláken liší hlavně v době běhu výpočtu. Z hlediska počtu epizod a dosažených odměn v jednotlivých epizodách můžeme pozorovat, že jsou lehce horší, ale víceméně na velice podobné úrovni. Můžeme však pozorovat, že samotná doba výpočtu se s přidáváním vláken u jednotlivých uzlů nejdříve snižuje a poté se od určitého počtu zase zvyšuje. Zároveň, díky možnosti přidáváním výpočetních *Slave* uzlů se doba běhu výpočtu zase zkracuje, ale také do určitého počtu těchto uzlů, kdy se potom jak doba výpočtu, tak i kvalita učení začíná zhoršovat. K zhoršování výsledků dochází opět stejně jako u typů učení *EPISODES* kvůli menší zpracovávané dávce zkušeností, jenž je ještě v případě toho typu učení menší, jelikož se proces učení zpracovává častěji než právě u typu učení *EPISODES*. Přidělována dávka zkušeností jednotlivým *Slave* uzlům

se získává zase dělením celkové paměti zkušeností, tedy z *Replay memory*, jenž obsahuje získané zkušeností za jeden cyklus prostředí. Právě pak s větším počtem *Slave* uzlů je tato paměť více dělena na menší dávky a potom tato dávka v může ještě být dělena v rámci tohoto uzlu pro jednotlivá vlákna. Tímto způsobem se pak zase právě stává více časové náročné samotné spouštění těchto vláken s čím dál menší přidělenou dávkou, než přímo sekvenční zpracování této dávky. Následně pak je také s větším počtem použitých uzlů *Slave* časově náročnější a větší problém se synchronizací dat ze všech těchto uzlů, kdy se právě při sloučení tato DQN data průměrují a tak určitým způsobem ztrácí svou kvalitu. Zase však z hlediska dosahované konečné odměny za epizodu vidíme, že všichni agenti se s využitím své DQN také úspěšně naučí operovat v daném prostředí.

Pak v případě co nejefektivnější konfigurace vychází nejlépe použití 3 až 4 *Slave* uzlů s jedním vláknem, jak je vidět z předchozích tabulek naměřených hodnot a grafu dosahované odměny za epizodu.

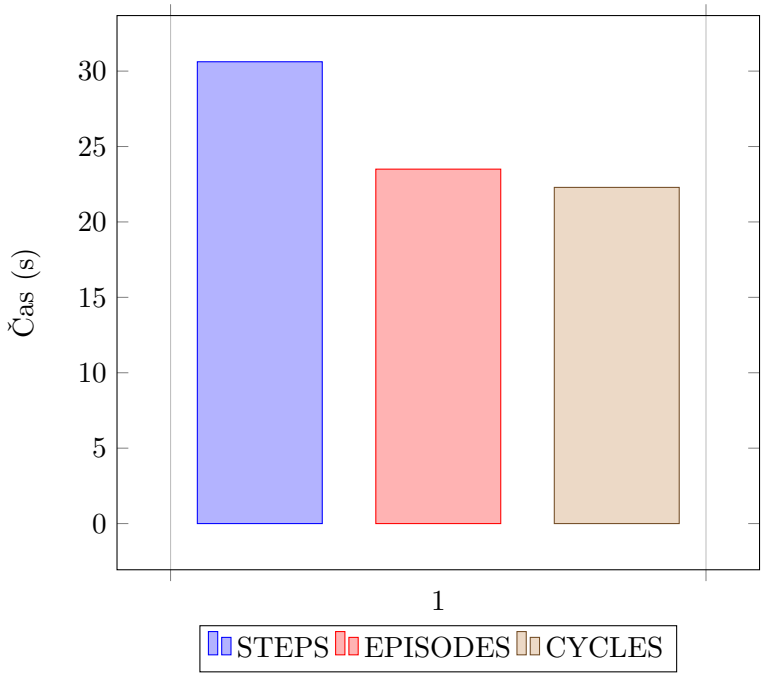
5.2.6 Vzájemné porovnání zvolených typů učení

Porovnávání všech zvolených typů učení mezi sebou není zcela jednoduché, jelikož každý typ učení může pracovat s daným způsobem paralelizace. Avšak dané typy učení využívající stejné nebo co nejvíce podobné způsoby paralelizace je možné porovnat, a pak lze porovnat mezi sebou nejlepší dosažené výsledky paralelizovaných přístupů jednotlivých typů učení.

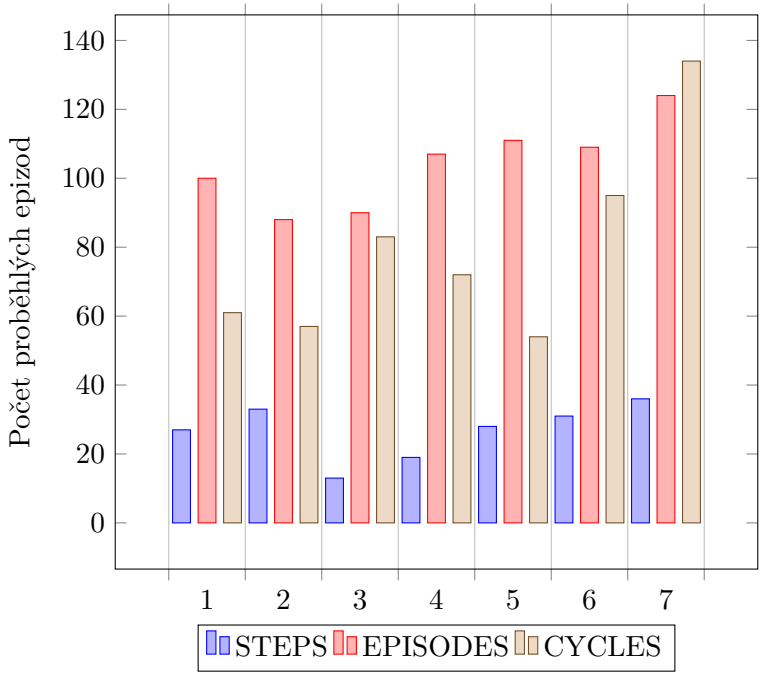
5.2.7 Porovnání v rámci vláken a velikosti dávky zkušeností

S využitím vláken lze tedy porovnávat typy učení *EPISODES* a *CYCLES*, ale pro účely porovnání určité podobné úrovní zahrneme i typ učení *STEPS* s danou velikostí dávky.

Následující grafy zobrazují dobu běhu výpočtu a počet proběhlých epizod prostředí těchto tří typů učení z hlediska podobnosti s využitím 1 vlákna pro typy učení *EPISODES* a *CYCLES* a s velikostí dávky 8 u *STEPS*. U grafu s porovnáním počtu epizod jsou postupně přidávány vlákna a zvyšována velikost dávky zkušeností.



Obrázek 5.9: Graf porovnání doby běhu zvolených typů učení s využitím 1 vlákna (EPISODES, CYCLES) a dávce zkušenosti o velikosti 8 (STEPS)



Obrázek 5.10: Graf porovnání počtu proběhlých epizod jednotlivých typů učení s postupným přidáváním počtu vláken (EPISODES, CYCLES) a zvyšováním dávky zkušeností (STEPS)

Z hlediska srovnání pomocí těchto grafů můžeme pozorovat, že nejkratší doby běhu výpočtu v

případě této základní konfigurace dosahuje typ učení *CYCLES* a nejdelší doby běhu pak typ učení *STEPS*.

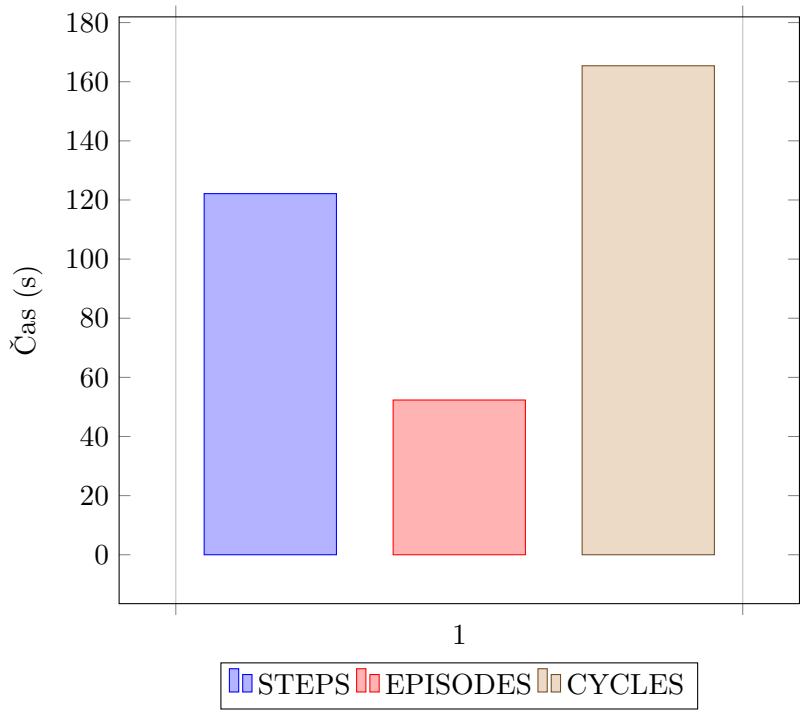
Naopak, zase u porovnání počtu proběhlých epizod, což jeden z faktorů značících kvalitu učení, můžeme vidět, že nejlepších výsledků dosahuje typ učení *STEPS* v případě nejmenších počtu potřebných epizod a v případě největšího počtu epizod pak typ učení *EPISODES*.

5.2.8 Porovnání paralelních modelů

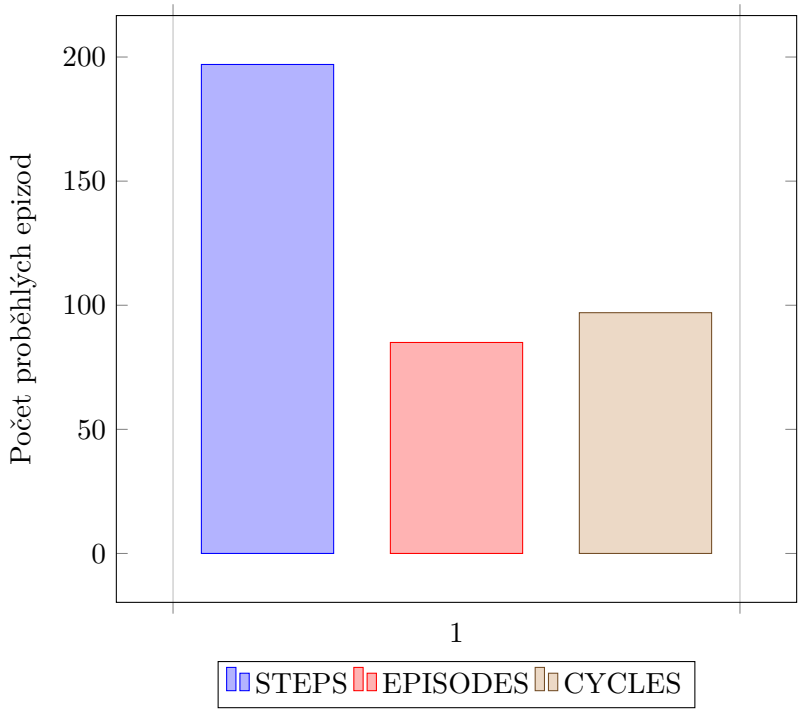
Pro účely tohoto porovnání typů učení s využitím paralelních modelů *Master-Slave* a *Gorilla DQN*, byly z důvodu velkého množství možných konfigurací, naměřených hodnot a nejednotnosti přístupu k paralelizaci vybrány konfigurace s nejlepšími dosahovanými výsledky.

Pro model *Master-Slave* byly nakonec pro dané typy učení vybrány stejné typy konfigurací z hlediska naměřených hodnot. Pro oba typy učení *EPISODES* a *CYCLES* byla tedy vybrána konfigurace s 3 *Slave* uzly a jedním použitým vláknem v rámci uzlu. Pro model *Gorilla DQN* a typ učení *STEPS* pak byla vybrána nejlepší konfigurace s 3 *Actor* uzly a 3 *Learner* uzly a dávkou zkušeností o velikostí 512.

Následující grafy zobrazují porovnání těchto paralelních modelu daných typů učení s předchozí popsanou konfigurací z hlediska doby běhu výpočtu a počtu proběhlých epizod, které určují kvalitu učení.



Obrázek 5.11: Graf porovnání doby běhu dostupných paralelních modelů a zvolených typů učení



Obrázek 5.12: Graf porovnání počtu proběhlých epizod dostupných paralelních modelů a jednotlivých typů učení

Z hlediska srovnání pomocí těchto grafů můžeme pozorovat, že nejkratší doby běhu výpočtu v případě této základní konfigurace dosahuje paralelní model *Master-Slave* s typem učení *EPISODES* a nejdelší doby běhu pak také tento model s typem učení *CYCLES*. Je to dáno především větší koncentrací síťové komunikace mezi výpočetními uzly u tohoto typu učení.

Pak z hlediska porovnání počtu proběhlých epizod, což jeden z faktorů značících kvalitu učení, můžeme vidět, že nejlepších výsledků dosahuje zase paralelní model *Master-Slave* s typem učení *EPISODES* v případě nejmenšího počtu potřebných epizod a v případě největšího počtu potřebných epizod pak paralelní model *Gorilla DQN* s typem učení *STEPS*.

Celkově v případě tohoto porovnání se tak nejlepších výsledků dosahuje paralelní model *Master-Slave* s typem učení *EPISODES*.

5.2.9 Celkové zhodnocení testování

Shrnutím předchozího porovnání můžeme usoudit, že lepších výsledků dosahuje paralelizace pomocí vláken oproti paralelním modelům *Master-Slave* a *Gorilla DQN*. Je to dáno hlavně kvůli rychlejší komunikaci mezi vlákny, než v případě síťové komunikace mezi výpočetními uzly a pak kvůli komplexnosti daných paralelních modelů.

V následující tabulce jsou pak uvedeny nejlepší konfigurace a výsledky jednotlivých paralelních přístupů získané z předchozích naměřených hodnot.

Tabulka 5.12: Nejlepší konfigurace a výsledky jednotlivých paralelních přístupů

Způsob paralelizace	Nejlepší konfigurace	Doba běhu (s)	Počet epizod
Vlákna	CYCLES, 4 vlákna	14,93	63
Velikost dávky zk.	STEPS, dávka 32	94,60	13
Master-Slave	EPISODES, 3 Slave uzly, 1 vlákno	52,34	85
GorillaDQN	STEPS, 3 Actor + 3 Learner uzly, dávka 512	116,99	197

Kapitola 6

Závěr

Tato práce se zabývala typem strojového učení Reinforcement learning se zaměřením na Deep Q-learning algoritmus, kdy se jedná o propojení Q-learning algorimu a hluboké umělé neuronové sítě.

V této práci bylo tedy nejdříve představeno strojové učení typu Reinforcement learning, poté byly představeny umělé neuronové sítě a nakonec byl představen a více popsán samotný vybraný algoritmus propojující tyto dvě oblasti.

Pro potřeby tohoto algoritmu muselo být z hlediska principů Reinforcement learning učení vytvořeno testovací stochastické prostředí, kde je možné algoritmus využít v rámci popsaného zavedeného rozhraní Agent-Environment. Jako podklad pro toto testovací prostředí byly vybrány arkádové hry od společnost Atari - Breakout a Pong. Dále bylo zadáním také prozkoumat možnosti a způsoby paralelizace tohoto algoritmu, jejich implementace pro možnosti spuštění na distribuovaných výpočetních uzlech a následné porovnání naměřených výsledků.

Následně byl popsán návrh celkového algoritmu, vybraného testovacího prostředí a jejich vzájemného propojení pro účely testování. Dále pak byly navrženy přístupy a architektury modelů pro možnosti paralelizace, kdy bylo vycházeno z již zavedených možných přístupů k řešení paralelizace hlavně v případě umělých neuronových sítí s pomocí distribuovaného přístupu, které pak byly uzpůsobeny pro použití s navrženým konceptem této práce. Pro realizaci tohoto návrhu pak bylo nutné také zvolit kompatibilní technologie pro vytvoření všech potřebných navržených konceptů.

Tento návrh byl pak tedy úspěšně postupně implementován. Následně byl tento implementovaný artefakt úspěšně otestován s různými typy konfigurací v testovacím prostředí a poté z hlediska paralelizace i na distribuovaných výpočetních uzlech IT4Innovations. Toto testování s různými konfiguracemi přineslo hlavně výsledky toho, jaké jsou limity a nejlepší podmínky pro průběh tohoto typu učení.

Výsledkem této práce je tedy úspěšná demonstrace Deep Q-learning algoritmu. Tato demonstrace je provedena s pomocí agenta pohybujícího se ve zvoleném testovacím prostředí arkádových her. Ten v tomto prostředí volí a vykonává dané akce a dokáže se tedy s využitím implementovaného Deep Q-learning algoritmu postupně naučit tyto hry úspěšně hrát a dosahovat dobrých výsledků.

Všechny vytvořené artefakty byly pak nakonec tedy přidány jako modul do již existujícího projektu NeuronNetModeler.

Literatura

1. SUTTON, Richard S.; BARTO, Andrew G. *Reinforcement learning: An Introduction*. Second edition. Cambridge: MA: The MIT Press, 2018. ISBN 978-0262039246.
2. *Markov Decision Processes* [online] [cit. 2021-01-24]. Dostupné z: `https://www.sciencedirect.com/topics/computer-science/markov-decision-process`.
3. *Epsilon-Greedy Algorithm in Reinforcement Learning* [online] [cit. 2021-01-24]. Dostupné z: `https://www.geeksforgeeks.org/epsilon-greedy-algorithm-in-reinforcement-learning/`.
4. *Reinforcement Learning algorithms — an intuitive overview* [online] [cit. 2021-01-23]. Dostupné z: `https://medium.com/@SmartLabAI/reinforcement-learning-algorithms-an-intuitive-overview-904e2dff5bbc`.
5. *Introduction to Various Reinforcement Learning Algorithms. Part I (Q-Learning, SARSA, DQN, DDPG)* [online] [cit. 2021-01-24]. Dostupné z: `https://towardsdatascience.com/introduction-to-various-reinforcement-learning-algorithms-i-q-learning-sarsa-dqn-ddpg-72a5e0cb6287`.
6. GOODFELLOW Ian, Yoshua BENGIO a Aaron COURVILLE. *Deep learning*. Cambridge: MA: MIT press, 2016. ISBN 978-0262035613.
7. SOARES; SOUZA, Alan M. *Neural Network Programming with Java*. Birmingham: UK: Packt Publishing, 2016. ISBN 978-1785880902.
8. *Feedforward Neural Networks* [online] [cit. 2021-01-22]. Dostupné z: `https://brilliant.org/wiki/feedforward-neural-networks/`.
9. *Deep Q-Network (DQN)-II* [online] [cit. 2021-01-23]. Dostupné z: `https://towardsdatascience.com/deep-q-network-dqn-ii-b6bf911b6b2c`.
10. *A Hands-On Introduction to Deep Q-Learning* [online] [cit. 2021-01-23]. Dostupné z: `https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/`.
11. *Deep Reinforcement Learning. Introduction. Deep Q Network (DQN) algorithm.* [Online] [cit. 2021-01-23]. Dostupné z: `https://medium.com/@markus.x.buchholz/deep-reinforcement-learning-introduction-deep-q-network-dqn-algorithm-fb74bf4d6862`.

12. *Atari* [online] [cit. 2021-02-20]. Dostupné z: https://www.atari.com/about-us/.

13. *Breakout* [online] [cit. 2021-02-20]. Dostupné z: https://www.classicarcademuseum.org/breakout.

14. *Pong* [online] [cit. 2021-02-20]. Dostupné z: https://www.britannica.com/topic/Pong.

15. *Scalable Deep Learning on Parallel and Distributed Infrastructures* [online] [cit. 2021-03-22]. Dostupné z: https://towardsdatascience.com/scalable-deep-learning-on-parallel-and-distributed-infrastructures-e5fb4a956bef.

16. *Java* [online] [cit. 2021-02-20]. Dostupné z: https://www.java.com/en/download/help/whatis_java.html.

17. *JavaFX* [online] [cit. 2021-02-20]. Dostupné z: https://openjfx.io/.

18. *Apache Maven* [online] [cit. 2021-02-20]. Dostupné z: https://maven.apache.org/what-is-maven.html.

19. *Aeron* [online] [cit. 2021-02-20]. Dostupné z: https://github.com/real-logic/aeron.

20. ING. IVO VONDRÁK, CSc. prof. *Neuronové sítě*. Ostrava: VŠB-TUO, 2001. ISBN 80-7078-259-5.

21. *IT4Innovations* [online] [cit. 2021-04-07]. Dostupné z: https://www.it4i.cz/en/for-users/documentation-and-support.